CORBAservices: Common Object Services Specification

Revised Edition - March 31, 1995
Updated: March 28, 1996
Updated: July 15, 1996
Updated: November 22, 1996
Updated: March 1997
Updated: July 1997
Updated: November 1997
Updated: December 1998

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Table of Contents

Preface . . . . **xlvii**

0.1 About This Document xlvii

0.1.1 Object Management Group xlvii

0.1.2 X/Open xlviii

0.2 Intended Audience xlviii

0.3 Need for Object Services xlviii

0.3.1 What Is an Object Service Specification? xlix

0.4 Associated Documents xlix

0.5 Structure of this Manual 1

0.6 Acknowledgements 1

1. Overview **1-1**

1.1 Summary of Key Features 1-1

1.1.1 Naming Service 1-1

1.1.2 Event Service 1-2

1.1.3 Life Cycle Service 1-2

1.1.4 Persistent Object Service 1-3

1.1.5 Transaction Service. 1-3

1.1.6 Concurrency Control Service 1-3

1.1.7 Relationship Service 1-4

1.1.8 Externalization Service 1-4

1.1.9 Query Service 1-5

1.1.10 Licensing Service 1-5

1.1.11 Property Service 1-5

1.1.12 Time Service. 1-6

1.1.13 Security Service 1-6

Contents

1.1.14 Object Trader Service 1-7
1.1.15 Object Collections Service 1-7

2. General Design Principles. 2-1

2.1 Service Design Principles. 2-1
2.1.1 Build on CORBA Concepts 2-1
2.1.2 Basic, Flexible Services 2-2
2.1.3 Generic Services. 2-2
2.1.4 Allow Local and Remote Implementations. . . . 2-2
2.1.5 Quality of Service is an Implementation Characteristic 2-2
2.1.6 Objects Often Conspire in a Service. 2-2
2.1.7 Use of Callback Interfaces 2-4
2.1.8 Assume No Global Identifier Spaces 2-4
2.1.9 Finding a Service is Orthogonal to Using It . . . 2-4

2.2 Interface Style Consistency 2-4
2.2.1 Use of Exceptions and Return Codes 2-4
2.2.2 Explicit Versus Implicit Operations 2-5
2.2.3 Use of Interface Inheritance 2-5

2.3 Key Design Decisions 2-5
2.3.1 Naming Service: Distinct from Property and Trading Services. 2-5
2.3.2 Universal Object Identity 2-5

2.4 Integration with Future Object Services. 2-5
2.4.1 Archive Service 2-6
2.4.2 Backup/Restore Service 2-6
2.4.3 Change Management Service 2-6
2.4.4 Data Interchange Service 2-6
2.4.5 Internationalization Service 2-6
2.4.6 Implementation Repository. 2-7
2.4.7 Interface Repository 2-7
2.4.8 Logging Service 2-7
2.4.9 Recovery Service 2-8
2.4.10 Replication Service. 2-8
2.4.11 Startup Service 2-9
2.4.12 Data Interchange Service 2-9

2.5 Service Dependencies. 2-9
2.5.1 Event Service 2-9
2.5.2 Life Cycle Service 2-9
2.5.3 Persistent Object Service 2-9
2.5.4 Relationship Service. 2-10

2.5.5 Externalization Service 2-10
2.5.6 Transaction Service 2-10
2.5.7 Concurrency Control Service 2-11
2.5.8 Query Service 2-11
2.5.9 Licensing Service 2-11
2.5.10 Property Service 2-12
2.5.11 Time Service 2-12
2.5.12 Security Service 2-12
2.5.13 Trader Service 2-12
2.5.14 Collections Service 2-12

2.6 Relationship to CORBA 2-12

2.6.1 ORB Interoperability Considerations: Transaction Service 2-12
2.6.2 Life Cycle Service 2-13
2.6.3 Naming Service 2-13
2.6.4 Relationship Service 2-13
2.6.5 Persistent Object Service 2-13
2.6.6 General Interoperability Requirements 2-13

2.7 Relationship to Object Model 2-14

2.8 Conformance to Existing Standards 2-14

3. Naming Service Specification 3-1

3.1 Service Description 3-1

3.1.1 Overview 3-1
3.1.2 Names 3-2
3.1.3 Names Library 3-3
3.1.4 Example Scenarios 3-3
3.1.5 Design Principles 3-4
3.1.6 Resolution of Technical Issues 3-5

3.2 The CosNaming Module 3-6

3.2.1 Binding Objects 3-8
3.2.2 Resolving Names 3-9
3.2.3 Unbinding Names 3-10
3.2.4 Creating Naming Contexts 3-11
3.2.5 Deleting Contexts 3-11
3.2.6 Listing a Naming Context 3-12
3.2.7 The BindingIterator Interface 3-12

3.3 The Names Library 3-13

3.3.1 Creating a Library Name Component 3-14
3.3.2 Creating a Library Name 3-15

Contents

3.3.3 The LNameComponent Interface 3-15

3.3.4 The LName Interface 3-15

3.3.4.1 Destroying a Library Name Component Pseudo Object 3-16

3.3.4.2 Inserting a Name Component. 3-16

3.3.4.3 Getting the i^{th} Name Component 3-16

3.3.4.4 Deleting a Name Component 3-17

3.3.4.5 Number of Name Components 3-17

3.3.4.6 Testing for Equality 3-17

3.3.4.7 Testing for Order 3-17

3.3.4.8 Producing an IDL form 3-18

3.3.4.9 Translating an IDL Form 3-18

3.3.4.10 Destroying a Library Name Pseudo-Object 3-18

4. Event Service Specification 4-1

4.1 Service Description 4-1

4.1.1 Overview 4-1

4.1.2 Event Communication 4-2

4.1.3 Example Scenario 4-2

4.1.4 Design Principles 4-4

4.1.5 Resolution of Technical Issues 4-4

4.1.6 Quality of Service. 4-6

4.2 Generic Event Communication. 4-6

4.2.1 Push Model. 4-6

4.2.2 Pull Model 4-7

4.3 The CosEventComm Module 4-8

4.3.1 The PushConsumer Interface 4-8

4.3.2 The PushSupplier Interface. 4-9

4.3.3 The PullSupplier Interface 4-9

4.3.4 The PullConsumer Interface 4-10

4.4 Event Channels. 4-10

4.4.1 Push-Style Communication with an Event Channel. 4-10

4.4.2 Pull-Style Communication with an Event Channel. 4-11

4.4.3 Mixed Style Communication with an Event Channel. 4-11

4.4.4 Multiple Consumers and Multiple Suppliers . . 4-12

4.4.5 Event Channel Administration 4-13

4.5 The CosEventChannelAdmin Module 4-15

4.5.1 The EventChannel Interface 4-16

4.5.2 The ConsumerAdmin Interface. 4-17

4.5.3 The SupplierAdmin Interface 4-17

4.5.4 The ProxyPushConsumer Interface. 4-17

4.5.5 The ProxyPullSupplier Interface. 4-18
4.5.6 The ProxyPullConsumer Interface 4-18
4.5.7 The ProxyPushSupplier Interface 4-19

4.6 Typed Event Communication 4-19
4.6.1 Typed Push Model 4-19
4.6.2 Typed Pull Model 4-20

4.7 The CosTypedEventComm Module 4-21
4.7.1 The TypedPushConsumer Interface 4-22
4.7.2 The TypedPullSupplier Interface 4-23

4.8 Typed Event Channels 4-23

4.9 The CosTypedEventChannelAdmin Module 4-24
4.9.1 The TypedEventChannel Interface 4-26
4.9.2 The TypedConsumerAdmin Interface. 4-26
4.9.3 The TypedSupplierAdmin Interface 4-27
4.9.4 The TypedProxyPushConsumer Interface. 4-27
4.9.5 The TypedProxyPullSupplier Interface. 4-28

4.10 Composing Event Channels and Filtering 4-28

4.11 Policies for Finding Event Channels 4-28

5. Persistent Object Service Specification 5-1

5.1 Introduction 5-1

5.2 Goals and Properties. 5-3
5.2.1 Basic Capabilities 5-3
5.2.2 Object-oriented Storage 5-3
5.2.2.1 Interfaces to Data 5-4
5.2.2.2 Self-description 5-4
5.2.2.3 Abstraction 5-4
5.2.3 Open Architecture. 5-4
5.2.4 Views of Service 5-5
5.2.4.1 Client 5-5
5.2.4.2 Object Implementation 5-6
5.2.4.3 Persistent Data Service 5-6
5.2.4.4 Datastore 5-6

5.3 Service Structure 5-7

5.4 The CosPersistencePID Module. 5-8
5.4.1 PID Interface 5-9
5.4.2 Example PIDFactory Interface 5-11

5.5 The CosPersistencePO Module 5-11
5.5.1 The PO Interface. 5-12
5.5.2 The POFactory Interface. 5-14
5.5.3 The SD Interface. 5-14

Contents

5.6 The CosPersistencePOM Module. 5-15
5.7 Persistent Data Service (PDS) Overview 5-18
5.8 The CosPersistencePDS Module 5-19
5.9 The Direct Access (PDS_DA) Protocol 5-21
5.10 The CosPersistencePDS_DA Module. 5-21
5.10.1 The PID_DA Interface 5-23
5.10.2 The Generic DAObject Interface 5-24
5.10.3 The DAObjectFactory Interface 5-24
5.10.4 The DAObjectFactoryFinder Interface 5-25
5.10.5 The PDS_DA Interface. 5-25
5.10.6 Defining and Using DA Data Objects. 5-26
5.10.7 The DynamicAttributeAccess Interface 5-28
5.10.8 The PDS_ClusteredDA Interface 5-29
5.11 The ODMG-93 Protocol. 5-30
5.12 The Dynamic Data Object (DDO) Protocol 5-30
5.13 The CosPersistenceDDO Module. 5-31
5.14 Other Protocols. 5-33
5.15 Datastores: CosPersistenceDS_CLI Module 5-34
5.15.1 The UserEnvironment Interface 5-36
5.15.2 The Connection Interface 5-37
5.15.3 The ConnectionFactory Interface 5-37
5.15.4 The Cursor Interface. 5-38
5.15.5 The CursorFactory Interface. 5-38
5.15.6 The PID_CLI Interface. 5-38
5.15.7 The Datastore_CLI Interface 5-40
5.16 Other Datastores. 5-43
5.17 Standards Conformance 5-43
5.18 References 5-43

6. Life Cycle Service Specification 6-1

6.1 Service Description 6-1
6.1.1 Overview 6-1
6.1.2 Organization of this Chapter. 6-3
6.1.3 Client’s Model of Object Life Cycle. 6-4
6.1.3.1 Client’s Model of Creation 6-4
6.1.3.2 Client’s Model of Deleting an Object . . . 6-6
6.1.3.3 Client’s Model of Copying or Moving an Object 6-6
6.1.4 Factory Finders. 6-7
6.1.4.1 Multiple Factory Finders 6-8
6.1.5 Design Principles 6-8

6.1.6 Resolution of Technical Issues 6-9
6.2 The CosLifeCycle Module 6-10
6.2.1 The LifeCycleObject Interface 6-11
6.2.1.1 copy 6-11
6.2.1.2 move 6-12
6.2.1.3 remove 6-13
6.2.2 The FactoryFinder Interface 6-13
6.2.2.1 find_factories 6-13
6.2.3 The GenericFactory Interface 6-14
6.2.3.1 create_object 6-15
6.2.3.2 supports 6-16
6.2.4 Criteria 6-17
6.3 Implementing Factories 6-18
6.3.1 Minimal Factories 6-19
6.3.2 Administered Factories 6-19
6.4 Target's Use of Factories and Factory Finders 6-21
6.5 Summary of Life Cycle Service 6-21
6.5.1 Summary of Life Cycle Service Structure 6-22
6.5.1.1 Creating a Compound Life Cycle Operation 6-33
6.5.1.2 Applying the Copy Operation to a Graph of Related Objects 6-33
6.5.1.3 Applying the Move Operation to a Graph of Related Objects 6-34
6.5.1.4 Applying the Remove Operation to a Graph of Related Objects 6-34
6.5.1.5 Destroying the Compound Operation. . . . 6-35
6.5.1.6 Copying a Node 6-35
6.5.1.7 Moving a Node 6-36
6.5.1.8 Removing a Node. 6-37
6.5.1.9 Getting the Node's Life Cycle Object . . . 6-37
6.5.1.10 Copying a Role 6-38
6.5.1.11 Moving a Role 6-38
6.5.1.12 Getting a Propagation Value 6-39
6.5.1.13 Copying the Relationship 6-39
6.5.1.14 Moving the Relationship 6-40
6.5.1.15 Getting a Propagation Value 6-41
6.5.1.16 Create a Traversal Criteria Based on Life Cycle Propagation Values 6-41
6.5.1.17 Visibility of the Federation Graph 6-52
6.5.1.18 Service Interface vs. Administration Interface 6-53
6.5.1.19 Multiple Service Interfaces 6-53
6.5.1.20 Cycles and Peer-to-Peer Relationships . 6-53
6.5.1.21 bind_generic_factory 6-56
6.5.1.22 unbind_generic_factory 6-56
6.5.1.23 resolve_generic_factory 6-57
6.5.1.24 list_generic_factories 6-57
6.5.1.25 match_service 6-57
6.5.1.26 get_hint. 6-57
6.5.1.27 get_link_properties 6-57

Contents

7. Concurrency Control Service 7-1

7.1 Service Description 7-1

7.1.1 Basic Concepts of Concurrency Control. 7-1

7.1.1.1 Clients and Resources 7-1

7.1.1.2 Transactions as Clients. 7-2

7.1.1.3 Locks 7-2

7.1.1.4 Lock Modes 7-2

7.1.1.5 Lock Granularity 7-2

7.1.1.6 Conflict Resolution 7-3

7.1.1.7 Conflict Resolution for Transactions . . . 7-3

7.1.1.8 Lock Duration. 7-3

7.2 Locking Model 7-3

7.2.1 Lock Modes 7-4

7.2.1.1 Read, Write, and Upgrade Locks 7-4

7.2.1.2 Intention Read and Intention Write Locks 7-4

7.2.1.3 Lock Mode Compatibility 7-5

7.2.2 Multiple Possession Semantics 7-5

7.3 Two-Phase Transactional Locking 7-6

7.4 Nested Transactions 7-6

7.5 CosConcurrencyControl Module 7-7

7.5.1 Types and Exceptions 7-9

7.5.2 LockCoordinator Interface 7-9

7.5.3 LockSet Interface 7-10

7.5.4 TransactionalLockSet Interface 7-11

7.5.5 LockSetFactory Interface 7-13

8. Externalization Service Specification. 8-1

8.1 Service Description 8-2

8.2 Service Structure 8-2

8.2.1 Client's Model of Object Externalization 8-3

8.2.2 Stream's Model of Object Externalization 8-3

8.2.3 Object's Model of Externalization 8-4

8.2.4 Object's Model of Internalization 8-6

8.3 Object and Interface Hierarchies 8-8

8.4 Interface Summary 8-11

8.4.1 Externalization Service Architecture: Audience/Bearer Mapping 8-12

8.5 CosExternalization Module 8-13

8.5.1 StreamFactory Interface 8-14

8.5.1.1 Creating a Stream Object 8-14

8.5.2 FileStreamFactory Interface 8-14

8.5.2.1 Creating a Stream Objec Associated with a File. 8-14

8.5.3 Stream Interface 8-14

8.5.3.1 Externalizing an Object 8-14
8.5.3.2 Externalizing Groups of Objects 8-15
8.5.3.3 Completing Externalization 8-15
8.5.3.4 Internalizing an Object 8-15

8.6 CosStream Module 8-16
8.6.1 Standard Stream Data Format 8-19
8.6.2 The StreamIO Interface 8-20
8.6.3 The Streamable Interface 8-21
8.6.3.1 Writing the Object's State to a Stream . . 8-21
8.6.3.2 Reinitializing the Object's State from a Stream 8-22
8.6.4 The StreamableFactory Interface 8-22
8.6.4.1 Creating a Streamable Object 8-22
8.6.5 The Node Interface 8-23
8.6.5.1 Externalizing a Node 8-23
8.6.5.2 Internalizing a Node 8-23
8.6.6 The Role Interface 8-24
8.6.6.1 Externalizing a Role 8-24
8.6.6.2 Internalizing a Role 8-24
8.6.6.3 Getting a Propagation Value 8-24
8.6.7 The Relationship Interface 8-25
8.6.7.1 Externalizing the Relationship 8-25
8.6.7.2 Internalizing the Relationship 8-25
8.6.7.3 Getting a Propagation Value 8-25
8.6.8 The PropagationCriteriaFactory Interface 8-26
8.6.8.1 Create a Traversal Criteria Based on Externalization Propagation 8-26

8.7 Specific Externalization Relationships 8-26

8.8 The CosExternalizationContainment Module 8-27

8.9 The CosExternalizationReference Module 8-28

8.10 Standard Stream Data Format 8-29
8.10.1 OMG Externalized Object Data 8-30
8.10.2 Externalized Repeated Reference Data 8-31
8.10.3 Externalized NIL Data 8-31

8.11 References 8-31

9. Relationship Service Specification 9-1

9.1 Service Description 9-1
9.1.1 Key Features of the Relationship Service 9-2
9.1.2 The Relationship Service vs. CORBA Object References 9-3
9.1.2.1 Relationships that Are Multidirectional . 9-3
9.1.2.2 Relationships that Allow Third Party Manipulation 9-4
9.1.2.3 Traversals that Are Supported for Graphs of Related Objects 9-4

Contents

9.1.2.4 Relationships and Roles that Can Be Extended with Attributes and Behavior . 9-4
9.1.3 Resolution of Technical Issues 9-4
9.1.3.1 Modeling and Relationship Semantics . . 9-4
9.1.3.2 Managing Relationships 9-4
9.1.3.3 Constraining Relationships. 9-5
9.1.3.4 Referential Integrity 9-5
9.1.3.5 Relationships and Roles as First Class Objects 9-5
9.1.3.6 Different Models for Navigating and Constructing Relationships 9-5
9.1.3.7 Efficiency Considerations 9-6
9.2 Service Structure 9-7
9.2.1 Levels of Service 9-7
9.2.1.1 Level One: Base Relationships. 9-7
9.2.1.2 Level Two: Graphs of Related Objects . . 9-8
9.2.1.3 Level Three: Specific Relationships 9-9
9.2.2 Hierarchy of Relationship Interface 9-10
9.2.3 Hierarchy of Role Interface 9-10
9.2.4 Interface Summary 9-11
9.3 The Base Relationship Model 9-13
9.3.1 Relationship Attributes and Operations 9-14
9.3.1.1 Rationale 9-15
9.3.2 Higher Degree Relationships 9-15
9.3.2.1 Rationale 9-15
9.3.3 Operations 9-17
9.3.3.1 Creation 9-17
9.3.3.2 Navigation 9-18
9.3.3.3 Destruction 9-18
9.3.4 Consistency Constraints 9-18
9.3.5 Implementation Strategies 9-19
9.3.6 The CosObjectIdentity Module. 9-19
9.3.6.1 The IdentifiableObject Interface 9-20
9.3.6.2 constant_random_id 9-20
9.3.6.3 is_identical 9-20
9.3.7 The CosRelationships Module 9-20
9.3.7.1 Example of Containment Relationships . 9-23
9.3.7.2 The RelationshipFactory Interface 9-23
9.3.7.3 The Relationship Interface 9-25
9.3.7.4 Destroying a Relationship 9-26
9.3.7.5 The Role Interface 9-26
9.3.7.6 The RoleFactory Interface 9-30
9.3.7.7 The RelationshipIterator Interface 9-32
9.4 Graphs of Related Objects 9-33
9.4.1 Graph Architecture 9-33
9.4.1.1 Nodes 9-35
9.4.2 Traversing Graphs of Related Objects 9-35
9.4.2.1 Detecting and Representing Cycles 9-35

9.4.2.2 Determining the Relevant Nodes and Edges 9-36
9.4.3 Compound Operations 9-36
9.4.4 An Example Traversal Criteria 9-37
9.4.4.1 Propagation 9-37
9.4.5 The CosGraphs Module 9-38
9.4.5.1 The TraversalFactory Interface 9-41
9.4.5.2 The Traversal Interface 9-42
9.4.5.3 The TraversalCriteria Interface 9-43
9.4.5.4 The Node Interface 9-44
9.4.5.5 The NodeFactory Interface 9-46
9.4.5.6 The Role Interface 9-46
9.4.5.7 The EdgeIterator Interface 9-46
9.5 Specific Relationships 9-47
9.5.1 Containment and Reference 9-47
9.5.2 The CosContainment Module 9-48
9.5.3 The CosReference Module 9-49
9.6 References 9-51

10. Transaction Service Specification 10-1

10.1 Service Description 10-2
10.1.1 Overview of Transactions 10-2
10.1.2 Transactional Applications 10-3
10.1.3 Definitions 10-3
10.1.3.1 Transactional Client 10-4
10.1.3.2 Transactional Object 10-4
10.1.3.3 Recoverable Objects and Resource Objects 10-5
10.1.3.4 Transactional Server 10-6
10.1.3.5 Recoverable Server 10-6
10.1.4 Transaction Service Functionality 10-6
10.1.4.1 Transaction Models 10-6
10.1.4.2 Transaction Termination 10-7
10.1.4.3 Transaction Integrity 10-8
10.1.4.4 Transaction Context 10-8
10.1.4.5 Synchronization 10-8
10.1.5 Principles of Function, Design, and Performance 10-8
10.1.5.1 Functional Requirements 10-9
10.1.5.2 Design Requirements 10-10
10.1.5.3 Performance Requirements 10-11
10.2 Service Architecture 10-12
10.2.1 Typical Usage 10-13
10.2.2 Transaction Context 10-13
10.2.3 Context Management 10-14
10.2.4 Datatypes 10-15
10.2.5 Structures 10-15
10.2.6 Exceptions 10-16
10.2.6.1 Standard Exceptions 10-16

Contents

10.2.6.2 Heuristic Exceptions. 10-16
10.2.6.3 WRONG_TRANSACTION Exception 10-17
10.2.6.4 Other Exceptions 10-17

10.3 Transaction Service Interfaces 10-17

10.3.1 Current Interface. 10-18
10.3.1.1 begin 10-18
10.3.1.2 commit 10-19
10.3.1.3 rollback 10-19
10.3.1.4 rollback_only 10-19
10.3.1.5 get_status 10-20
10.3.1.6 get_transaction_name 10-20
10.3.1.7 set_timeout 10-20
10.3.1.8 get_control 10-20
10.3.1.9 suspend 10-20
10.3.1.10 resume 10-21

10.3.2 TransactionFactory Interface 10-21
10.3.2.1 create 10-21
10.3.2.2 parameter has a nonzero value *n*. 10-21
10.3.2.3 recreate 10-21

10.3.3 Control Interface. 10-22
10.3.3.1 get_terminator 10-22
10.3.3.2 get_coordinator 10-22

10.3.4 Terminator Interface 10-23
10.3.4.1 commit 10-23
10.3.4.2 rollback. 10-23

10.3.5 Coordinator Interface 10-24
10.3.5.1 get_status 10-25
10.3.5.2 get_parent_status 10-25
10.3.5.3 get_top_level_status 10-26
10.3.5.4 is_same_transaction 10-26
10.3.5.5 is_ancestor_transaction 10-26
10.3.5.6 is_descendant_transaction 10-26
10.3.5.7 is_related_transaction 10-26
10.3.5.8 is_top_level_transaction 10-26
10.3.5.9 hash_transaction 10-26
10.3.5.10 hash_top_level_tran 10-27
10.3.5.11 register_resource 10-27
10.3.5.12 register_synchronization 10-27
10.3.5.13 register_subtran_aware 10-27
10.3.5.14 rollback_only 10-28
10.3.5.15 get_transaction_name. 10-28
10.3.5.16 create_subtransaction 10-28
10.3.5.17 get_txcontext 10-28

10.3.6 Recovery Coordinator Interface 10-29
10.3.6.1 replay_completion 10-29

10.3.7 Resource Interface 10-29
10.3.7.1 prepare 10-30
10.3.7.2 rollback 10-31
10.3.7.3 commit 10-31
10.3.7.4 commit_one_phase 10-31
10.3.7.5 forget 10-32

10.3.8 Synchronization Interface. 10-32

10.3.8.1 before_completion 10-32
10.3.8.2 after_completion 10-32
10.3.9 Subtransaction Aware Resource Interface. 10-33
10.3.9.1 commit_subtransaction 10-33
10.3.9.2 rollback_subtransaction 10-33
10.3.10 TransactionalObject Interface 10-34
10.4 The User's View 10-34
10.4.1 Application Programming Models 10-34
10.4.1.1 Direct Context Management: Explicit Propagation 10-35
10.4.1.2 Indirect Context Management: Implicit Propagation 10-35
10.4.1.3 Indirect Context Management: Explicit Propagation 10-35
10.4.1.4 Direct Context Management: Implicit Propagation 10-35
10.4.2 Interfaces 10-36
10.4.3 Checked Transaction Behavior 10-36
10.4.4 X/Open Checked Transactions 10-37
10.4.4.1 Reply Check 10-37
10.4.4.2 Commit Check 10-37
10.4.4.3 Resume Check 10-38
10.4.5 Implementing a Transactional Client: Heuristic Completions 10-38
10.4.6 Implementing a Recoverable Server 10-38
10.4.6.1 Recoverable Object 10-38
10.4.6.2 Resource Object 10-39
10.4.6.3 Reliable Servers 10-39
10.4.7 Application Portability 10-39
10.4.7.1 Flat Transactions 10-39
10.4.7.2 X/Open Checked Transactions 10-39
10.4.8 Distributed Transactions 10-40
10.4.9 Applications Using Both Checked and Unchecked Services 10-40
10.4.10 Examples 10-40
10.4.10.1 A Transaction Originator: Indirect and Implicit 10-41
10.4.10.2 Transaction Originator: Direct and Explicit 10-41
10.4.10.3 Example of a Recoverable Server 10-42
10.4.10.4 Example of a Transactional Object . . . 10-43
10.4.11 Model Interoperability 10-44
10.4.11.1 Importing Transactions 10-44
10.4.11.2 Exporting Transactions 10-46
10.4.11.3 Programming Rules 10-46
10.4.12 Failure Models 10-47
10.4.12.1 Transaction Originator 10-47
10.4.12.2 Transactional Server 10-48
10.4.12.3 Recoverable Server 10-48

10.5 The Implementers' View 10-48

10.5.1 Transaction Service Protocols. 10-49

10.5.1.1 General Principles 10-49

10.5.1.2 Normal Transaction Completion 10-50

10.5.1.3 Failures and Recovery 10-56

10.5.1.4 Transaction Completion after Failure . . 10-57

10.5.2 ORB/TS Implementation Considerations 10-60

10.5.2.1 Transaction Propagation 10-60

10.5.2.2 Transaction Service Interoperation 10-61

10.5.2.3 Transaction Service Portability 10-63

10.5.3 Model Interoperability 10-67

10.6 The CosTransactions Module. 10-69

10.6.1 The CosTSPortability Module 10-73

10.6.1.1 tx_open 10-75

10.6.1.2 tx_close 10-75

10.6.1.3 tx_begin 10-75

10.6.1.4 tx_rollback 10-75

10.6.1.5 tx_commit and tx_set_commit_return . . 10-75

10.6.1.6 tx_set_transaction_control 10-75

10.6.1.7 tx_set_transaction_timeout. 10-76

10.6.1.8 tx_info 10-76

10.6.1.9 OSI TP Transaction Identifiers 10-81

10.6.1.10 Incoming OSI TP Communications (Imported Transactions) 10-81

10.6.1.11 Outgoing OSI TP Communications (Exported Transactions) 10-82

10.6.1.12 LU 6.2 Transaction Identifiers 10-82

10.6.1.13 Incoming LU 6.2 Communications . . . 10-83

10.6.1.14 Outgoing LU 6.2 Communications . . . 10-83

10.6.1.15 ODMG Standard 10-83

10.6.1.16 Integration of ODMG ODBMSs with the Transaction Service 10-84

11. Query Service Specification 11-1

11.1 Service Description 11-1

11.1.1 Overview 11-1

11.1.2 Design Principles 11-1

11.1.3 Architecture 11-2

11.1.3.1 Query Evaluators: Nesting and Federation 11-3

11.1.3.2 Collections 11-4

11.1.3.3 Queryable Collections for Scope and Result 11-5

11.1.3.4 Query Objects 11-5

11.1.4 Query Languages 11-6

11.1.4.1 SQL Query 11-7

11.1.4.2 OQL 11-7

11.1.4.3 SQL Query = OQL 11-8

11.1.5 Key Features 11-9

11.2 Service Structure 11-10

11.2.1 Overview 11-10

11.2.1.1 Type One: Collections 11-10
11.2.1.2 Type Two: Query Framework 11-10
11.2.2 Collection Interface Structure. 11-10
11.2.3 Query Framework Interface Hierarchy/ Structure 11-10
11.2.4 Interface Overview 11-11
11.3 The Collection Model. 11-12
11.3.1 Common Types of Collections 11-12
11.3.2 Iterators. 11-12
11.4 The CosQueryCollection Module. 11-14
11.4.1 The CollectionFactory Interface 11-15
11.4.1.1 Creating a Collection 11-15
11.4.2 The Collection Interface 11-16
11.4.2.1 Determining the Cardinality 11-16
11.4.2.2 Adding an Element 11-16
11.4.2.3 Adding Elements from a Collection . . . 11-17
11.4.2.4 Inserting an Element 11-17
11.4.2.5 Replacing an Element 11-17
11.4.2.6 Removing an Element 11-17
11.4.2.7 Removing all Elements 11-18
11.4.2.8 Retrieving an Element 11-18
11.4.2.9 Creating an Iterator 11-18
11.4.3 The Iterator Interface 11-18
11.4.3.1 Accessing the Current Element 11-18
11.4.3.2 Resetting the Iteration 11-18
11.4.3.3 Testing for Completion of an Iteration . 11-19
11.5 The Query Framework Model 11-19
11.5.1 Query Evaluators 11-19
11.5.2 Queryable Collections 11-20
11.5.3 Query Managers 11-21
11.5.4 Query Objects. 11-21
11.6 The CosQuery Module 11-23
11.6.1 The QueryLanguageType Interfaces 11-24
The QueryEvaluator Interface. 11-25
11.6.1.1 Determining the Supported Query Language Types 11-25
11.6.1.2 Determining the Default Query Language Type 11-25
11.6.1.3 Evaluating a Query 11-25
11.6.2 The QueryableCollection Interface. 11-25
11.6.3 The QueryManager Interface 11-25
11.6.3.1 Creating a Query Object 11-26
11.6.4 The Query Interface 11-26
11.6.4.1 Determining the Associated Query Manager 11-26
11.6.4.2 Preparing the Query for Execution 11-26
11.6.4.3 Executing the Query 11-26

Contents

11.6.4.4 Determining the Query Status 11-27
11.6.4.5 Obtaining the Query Result 11-27

11.7 References 11-27

12. Licensing Service Specification. 12-1

12.1 Existing License Management Products. 12-1
12.1.1 Business Policy. 12-2
12.1.2 License Types 12-2
12.1.3 A History of License Types 12-3
12.1.4 Asset Management 12-3
12.1.5 License Usage Practices 12-4
12.1.6 Scalability. 12-4
12.1.7 Reliability 12-4
12.1.8 Legacy Applications 12-5
12.1.9 Security. 12-6
12.1.10 Client/Server Authentication 12-6
12.1.11 Example: Application Acquiring and Releasing a Concurrent License. 12-6

12.2 Service Description 12-7
12.2.1 Overview 12-7
12.2.2 Key Components of a Licensing System 12-8
12.2.2.1 License Attributes 12-8
12.2.2.2 Licensing Policy 12-8
12.2.2.3 Interfaces Isolated From Business Policies 12-10
12.2.3 Licensing in the CORBA Environment. 12-10
12.2.4 Design Principles 12-12
12.2.5 Licensing Service Interfaces. 12-13
12.2.5.1 Interfaces are Mandatory 12-13
12.2.5.2 Constraints on Object Behavior 12-13
12.2.6 Licensing Event Trace Diagram 12-14

12.3 The CosLicensing Module 12-17
12.3.1 LicenseServiceManager Interface. 12-19
12.3.2 ProducerSpecificLicenseService Interface 12-20

12.4 References 12-22

13. Property Service 13-1

13.1 Overview 13-1
13.1.1 Service Description. 13-1
13.1.1.1 Client's Model of Properties 13-2
13.1.1.2 Object's Model of Properties 13-2
13.1.2 OMG IDL Interface Summary 13-3
13.1.3 Summary of Key Features. 13-3

13.2 Service Interfaces 13-4
13.2.1 CosPropertyService Module 13-4
13.2.1.1 Data Types 13-5
13.2.1.2 Exceptions 13-7
13.2.2 PropertySet Interface 13-9
13.2.2.1 Defining and Modifying Properties 13-9
13.2.2.2 define_properties 13-10
13.2.2.3 Listing and Getting Properties 13-10
13.2.2.4 get_all_property_names 13-11
13.2.2.5 get_property_value 13-11
13.2.2.6 get_properties 13-11
13.2.2.7 get_all_properties 13-12
13.2.2.8 Deleting Properties 13-12
13.2.2.9 delete_property 13-12
13.2.2.10 delete_properties 13-12
13.2.2.11 delete_all_properties 13-12
13.2.2.12 Determining If a Property Is Already Defined 13-13
13.2.3 PropertySetDef Interface 13-14
13.2.3.1 Retrieval of PropertySet Constraints . . . 13-14
13.2.3.2 get_allowed_properties 13-15
13.2.3.3 Defining and Modifying Properties with Modes 13-15
13.2.3.4 define_properties_with_modes 13-16
13.2.3.5 Getting and Setting Property Modes . . . 13-17
13.2.3.6 get_property_modes 13-18
13.2.3.7 set_property_mode 13-18
13.2.3.8 set_property_modes 13-18
13.2.4 PropertiesIterator Interface 13-18
13.2.4.1 next_one, next_n 13-19
13.2.4.2 Destroying the Iterator 13-19
13.2.5 PropertyNamesIterator Interface. 13-19
13.2.5.1 Resetting the Position in an Iterator . . . 13-20
13.2.5.2 next_one, next_n 13-20
13.2.5.3 Destroying the Iterator 13-20
13.2.6 PropertySetFactory Interface 13-20
13.2.7 PropertySetDefFactory Interface 13-21

14. Time Service Specification 14-1
14.1 Introduction 14-1
14.1.1 Time Service Requirements 14-1
14.1.2 Representation of Time. 14-1
14.1.3 Source of Time 14-2
14.1.4 General Object Model. 14-3
14.1.5 Conformance Points 14-4
14.2 Basic Time Service. 14-4
14.2.1 Object Model 14-4
14.2.2 Data Types 14-5
14.2.2.1 Type TimeT 14-6

Contents

14.2.2.2 Type InaccuracyT 14-6
14.2.2.3 Type TdfT 14-6
14.2.2.4 Type UtcT 14-6
14.2.2.5 Type IntervalT 14-6
14.2.2.6 Enum ComparisonType 14-7
14.2.2.7 Enum TimeComparison 14-7
14.2.2.8 Enum OverlapType 14-7

14.2.3 Exceptions 14-8
14.2.3.1 TimeUnavailable 14-8

14.2.4 Universal Time Object (UTO) 14-8
14.2.4.1 Readonly attribute time 14-9
14.2.4.2 Readonly attribute inaccuracy 14-9
14.2.4.3 Readonly attribute tdf 14-9
14.2.4.4 Readonly attribute utc_time 14-9
14.2.4.5 Operation absolute_time 14-9
14.2.4.6 Operation compare_time 14-10
14.2.4.7 Operation time_to_interval 14-10
14.2.4.8 Operation interval 14-10

14.2.5 Time Interval Object (TIO) 14-10
14.2.5.1 Readonly attribute time_interval 14-10
14.2.5.2 Operation spans 14-11
14.2.5.3 Operation overlaps 14-11
14.2.5.4 Operation time 14-11

14.2.6 Time Service 14-11
14.2.6.1 Operation universal_time 14-12
14.2.6.2 Operation secure_universal_time 14-12
14.2.6.3 Operation new_universal_time 14-12
14.2.6.4 Operation uto_from_utc 14-12
14.2.6.5 Operation new_interval 14-12

14.3 Timer Event Service 14-13

14.3.1 Object Model 14-13

14.3.2 Usage 14-13

14.3.3 Data Types 14-14
14.3.3.1 Enum TimeType 14-14
14.3.3.2 Enum EventStatus 14-14
14.3.3.3 Type TimerEventT 14-15

14.3.4 Exceptions 14-15

14.3.5 Timer Event Handler. 14-15
14.3.5.1 Attribute status 14-16
14.3.5.2 Operation time_set 14-16
14.3.5.3 Operation set_timer 14-16
14.3.5.4 Operation cancel_timer 14-16
14.3.5.5 Operation set_data 14-16

14.3.6 Timer Event Service 14-16
14.3.6.1 Operation register 14-17
14.3.6.2 Operation unregister 14-17
14.3.6.3 Operation event_time 14-17

14.4 Conformance 14-17

15. Security Service Specification 15-1

15.1 Introduction to Security 15-2

15.1.1 Why Security? 15-2

15.1.2 What Is Security? 15-3

15.1.3 Threats in a Distributed Object System 15-3

15.1.4 Summary of Key Security Features 15-4

15.1.5 Goals. 15-5

15.1.5.1 Simplicity 15-5
15.1.5.2 Consistency 15-5
15.1.5.3 Scalability 15-5
15.1.5.4 Usability for End Users 15-5
15.1.5.5 Usability for Administrators 15-6
15.1.5.6 Usability for Implementors 15-6
15.1.5.7 Flexibility of Security Policy 15-6
15.1.5.8 Independence of Security Technology . 15-6
15.1.5.9 Application Portability 15-7
15.1.5.10 Interoperability 15-7
15.1.5.11 Performance 15-7
15.1.5.12 Object Orientation 15-7
15.1.5.13 Specific Security Goals 15-8
15.1.5.14 Security Architecture Goals 15-8

15.2 Introduction to the Specification 15-9

15.2.1 Specification Structure 15-10

15.2.1.1 Structure of the Chapter 15-10
15.2.1.2 Normative and Non-normative Material 15-11
15.2.1.3 Section Summaries 15-11

15.2.2 CORBA Security and Secure Interoperability Feature Packages 15-12

15.2.2.1 Main Security Functionality Packages . 15-13
15.2.2.2 Optional Security Functionality Packages 15-13
15.2.2.3 Security Replaceability Packages 15-13
15.2.2.4 Common Secure Interoperability (CSI) Feature packages 15-14
15.2.2.5 SECIOP Interoperability package 15-15
15.2.2.6 Security Mechanism packages 15-15
15.2.2.7 SECIOP Plus DCE-CIOP Interoperability 15-16

15.2.3 Feature Packages and Modules. 15-17

15.3 Security Reference Model 15-18

15.3.1 Definition of a Security Reference Model 15-18

15.3.2 Principals and Their Security Attributes. 15-20

15.3.3 Secure Object Invocations 15-21

15.3.3.1 Establishing Security Associations 15-22
15.3.3.2 Message Protection 15-22

15.3.4 Access Control Model 15-24

15.3.4.1 Object Invocation Access Policy 15-25
15.3.4.2 Application Access Policy 15-26

Contents

15.3.4.3 Access Policies 15-26
15.3.4.4 Privilege Attributes 15-27
15.3.4.5 Control Attributes 15-27
15.3.4.6 Access Policies Supported by This Specification 15-28
15.3.5 Auditing 15-28
15.3.6 Delegation. 15-30
15.3.6.1 Privilege Delegation 15-31
15.3.6.2 Overview of Delegation Schemes 15-31
15.3.6.3 Facilities Potentially Available 15-32
15.3.6.4 Specifying Delegation Options 15-35
15.3.6.5 Technology Support for Delegation Options 15-35
15.3.7 Non-repudiation 15-35
15.3.8 Domains 15-38
15.3.8.1 Security Policy Domains 15-38
15.3.8.2 Security Environment Domains 15-41
15.3.8.3 Security Technology Domains 15-42
15.3.8.4 Domains and Interoperability 15-42
15.3.9 Security Management and Administration 15-44
15.3.9.1 Managing Security Policy Domains. . . . 15-44
15.3.9.2 Managing Security Environment Domains 15-44
15.3.9.3 Managing Security Technology Domains 15-45
15.3.10 Implementing the Model. 15-45
15.4 Security Architecture 15-45
15.4.1 Different Users' View of the Security Model . . 15-46
15.4.1.1 Enterprise Management View 15-46
15.4.1.2 End User's View 15-47
15.4.1.3 Application Developer's View 15-47
15.4.1.4 Administrator's View 15-48
15.4.1.5 Object System Implementor's View . . . 15-48
15.4.2 Structural Model. 15-49
15.4.2.1 Application Components 15-50
15.4.2.2 ORB Services 15-51
15.4.2.3 Security Services 15-53
15.4.2.4 Security Policies and Domain Objects . 15-53
15.4.3 Security Technology 15-55
15.4.4 Basic Protection and Communications 15-56
15.4.4.1 Environment Domains 15-56
15.4.4.2 Component Protection 15-56
15.4.5 Security Object Models 15-58
15.4.5.1 The Model as Seen by Applications . . . 15-58
15.4.5.2 Administrative Model 15-75
15.4.5.3 The Model as Seen by the Objects Implementing Security 15-79
15.4.5.4 Summary of Objects in the Model 15-87
15.5 Application Developer's Interfaces 15-88

15.5.1 Introduction 15-88
15.5.1.1 Security Functionality Packages 15-89
15.5.1.2 Introduction to the Interfaces 15-89
15.5.2 Finding Security Features 15-90
15.5.2.1 Description of Facilities 15-90
15.5.3 Authentication of Principals 15-90
15.5.3.1 Description of Facilities 15-90
15.5.3.2 The SecurityLevel2::Principal Authenticator Interface 15-91
15.5.3.3 Portability Implications 15-93
15.5.4 The Credentials Object 15-94
15.5.4.1 Description of Facilities 15-94
15.5.4.2 The SecurityLevel2::Credentials Interface 15-95
15.5.4.3 Portability Implications 15-99
15.5.5 Operations on Object Reference 15-99
15.5.5.1 Description of Facilities 15-99
15.5.5.2 Client-Side Invocation Policy Objects 15-101
15.5.5.3 Security-Relevant Operations in the CORBA::Object Interface 15-102
15.5.5.4 Portability Implications 15-104
15.5.6 Security Operations on Current 15-105
15.5.6.1 Description 15-105
15.5.6.2 The SecurityLevel1::Current Interface . 15-107
15.5.6.3 The SecurityLevel2::Current Interface . 15-108
15.5.7 Security Audit. 15-113
15.5.7.1 Description of Facilities 15-113
15.5.7.2 The SecurityLevel2::AuditDecision Interface 15-114
15.5.7.3 The SecurityLevel2::AuditChannel Interface 15-115
15.5.7.4 Portability Implications 15-116
15.5.8 Administering Security Policy 15-116
15.5.9 Access Control 15-117
15.5.9.1 Description of Facilities 15-117
15.5.9.2 The Access Decision Object 15-118
15.5.9.3 The SecurityLevel2::AccessDecision Interface 15-118
15.5.9.4 Portability Implications 15-119
15.5.10 Delegation Facilities 15-120
15.5.10.1 Description of Facilities 15-120
15.5.10.2 Operations 15-120
15.5.10.3 Portability Implications 15-121
15.5.11 Non-repudiation 15-121
15.5.11.1 Description of Facilities 15-121
15.5.11.2 Non-repudiation Service Data Types . 15-123
15.5.11.3 The NRservice::NRCredentials Interface 15-124
15.6 Administrator's Interfaces 15-130
15.6.1 Concepts 15-131

Contents

15.6.1.1 Administrators 15-131
15.6.1.2 Policy Domains 15-131
15.6.1.3 Security Policies 15-131
15.6.2 Domain Management 15-132
15.6.3 Security Policies Introduction. 15-132
15.6.4 Access Policies 15-133
15.6.4.1 Rights 15-133
15.6.4.2 The SecurityLevel2::RequiredRights Interface 15-134
15.6.4.3 The SecurityAdmin::AccessPolicy Interface 15-136
15.6.4.4 Specific Invocation Access Policies . . . 15-137
15.6.4.5 The Domain AccessPolicy Object 15-137
15.6.4.6 The SecurityAdmin::DomainAccessPolicy Interface 15-141
15.6.5 Audit Policies 15-144
15.6.5.1 The SecurityAdmin::AuditPolicy Interface 15-144
15.6.6 Secure Invocation and Delegation Policies . . 15-147
15.6.6.1 Secure Invocation Policies 15-148
15.6.6.2 Secure Association Options 15-149
15.6.6.3 The SecurityAdmin::SecureInvocation Policy Interface 15-150
15.6.6.4 The SecurityAdmin::Delegation Policy Interface 15-152
15.6.7 Non-repudiation Policy Management 15-153
15.6.7.1 Data Types for Non-repudiation Policy Management Interfaces 15-154
15.6.7.2 The NRservice::NRPolicy Interface . . . 15-154
15.7 Implementor’s Security Interfaces 15-156
15.7.1 Security Interceptors. 15-156
15.7.1.1 Invocation Time Policies 15-158
15.7.1.2 Secure Invocation Interceptor 15-158
15.7.1.3 Access Control Interceptor 15-161
15.7.2 Implementation-Level Security Object Interfaces 15-161
15.7.2.1 The Vault Object 15-162
15.7.2.2 The SecurityReplaceable::Vault Interface 15-162
15.7.2.3 The Security Context Object 15-165
15.7.2.4 The SecurityReplaceable::Security Context Interface 15-166
15.7.2.5 The Credentials Object 15-169
15.7.2.6 The Access Decision Object 15-169
15.7.2.7 Audit Objects 15-169
15.7.2.8 Principal Authentication 15-170
15.7.2.9 Non-repudiation 15-170
15.7.3 Replaceable Security Services 15-170
15.7.3.1 Replacing Authentication and Security Association Services 15-170
15.7.3.2 Replacing Access Control Policies 15-170

15.7.3.3 Replacing Audit Services 15-170
15.7.3.4 Replacing Non-repudiation Services . . . 15-171
15.7.3.5 Other Replaceability 15-171
15.7.3.6 Linking to External Security Services . . 15-171

15.8 Security Interoperability Protocols. 15-172

15.8.1 Introduction 15-172

15.8.2 Interoperability Model 15-173

15.8.2.1 Security Information in the Object Reference 15-174
15.8.2.2 Establishing a Security Association . . . 15-174
15.8.2.3 Protecting Messages 15-175
15.8.2.4 Security Mechanisms for Secure Object Invocations 15-175
15.8.2.5 Security Mechanism Types 15-175
15.8.2.6 Interoperating between Security Policy Domains 15-176
15.8.2.7 Secure Interoperability Bridges 15-177

15.8.3 Protocol Enhancements 15-178

15.8.4 CORBA Interoperable Object Reference with Security. 15-178

15.8.4.1 Security Components of the IOR 15-179
15.8.4.2 IOR Example 15-181
15.8.4.3 Operational Semantics 15-181

15.8.5 Common Secure Interoperability Levels. . . . 15-183

15.8.6 Key Distribution Types 15-184

15.8.7 Security Mechanisms Hosted on SECIOP . . 15-184

15.8.7.1 SPKM Protocol 15-184
15.8.7.2 GSS Kerberos Protocol 15-184
15.8.7.3 CSI-ECMA protocol 15-184

15.8.8 Security Mechanisms Hosted Directly on IIOP 15-185

15.8.9 Choices of Protocols, Cryptographic Profiles, and Key Technologies. 15-185

15.8.9.1 Choice of Protocol and Key Technology 15-185
15.8.9.2 Cryptographic Profiles 15-186
15.8.9.3 Conformance to External Security Mechanisms 15-186

15.8.10 Common Secure Interoperability Requirements 15-186

15.8.10.1 CORBA Standard Security Mechanisms 15-187
15.8.10.2 International Deployment 15-187
15.8.10.3 Consistency 15-188
15.8.10.4 Scalability 15-188
15.8.10.5 Flexibility of Security Policy 15-189
15.8.10.6 Application Portability 15-189
15.8.10.7 Security Services Portability/ Replaceability 15-190
15.8.10.8 Performance 15-190
15.8.10.9 Identifying Encumbered Technology . 15-190

Contents

15.8.11 Relation to CORBA Security Facilities and Interfaces 15-191
15.8.11.1 Functionality 15-191
15.8.11.2 Replaceability 15-191
15.8.11.3 Levels of Interoperability 15-192
15.8.12 Security Functionality. 15-194
15.8.12.1 Authentication 15-194
15.8.12.2 Access Control 15-194
15.8.12.3 Audit 15-194
15.8.12.4 Secure Invocation 15-194
15.8.12.5 Delegation Facilities 15-195
15.8.12.6 Non-repudiation 15-195
15.8.12.7 Security Policies 15-195
15.8.13 Model for Use and Contents of Credentials . 15-196
15.8.13.1 Credential Content at the Client 15-197
15.8.13.2 Attributes During Transmission 15-197
15.8.13.3 Attributes at the Target 15-198
15.8.13.4 Mapping Security Names to Externally Valid Identities 15-199
15.8.13.5 Mapping Other Attributes to Externally Valid IDL Attributes 15-199
15.8.13.6 Mapping to Local Attribute Values . . . 15-200
15.8.14 CORBA Interfaces 15-200
15.8.14.1 Service Options for Common Secure Interoperability 15-201
15.8.14.2 Mechanism Types 15-201
15.8.14.3 Delegation-Related Interfaces 15-202
15.8.15 Support for CORBA Security Facilities and Extensibility 15-202
15.8.16 Security Replaceability for ORB Security Implementors 15-203
15.8.16.1 Attribute Mapping 15-203
15.8.16.2 Use of GSS-API 15-203
15.9 Secure Inter-ORB Protocol (SECIOP) 15-204
15.9.1 Architectural Assumptions 15-205
15.9.2 SECIOP Sequencing Layer. 15-206
15.9.2.1 Protocol State 15-207
15.9.2.2 Protocol Initialization 15-209
15.9.2.3 Upon Receipt of a SequencedData Frame 15-210
15.9.2.4 Sending a SequencedDataFrame 15-211
15.9.3 SECIOP Context Management Layer 15-212
15.9.3.1 SECIOP Context Management Layer Message Header 15-212
15.9.3.2 SECIOP Context Management Layer Protocol 15-213
15.9.3.3 ContextId 15-213
15.9.3.4 ContextIdDefn 15-213
15.9.3.5 TokenType 15-214
15.9.3.6 Message Definitions 15-214

15.9.4 SECIOP Context Management Finite State Machine Tables 15-217
15.9.4.1 SECIOP Context Management Protocol State Tables 15-217
15.10 The SECIOP-Hosted CSI Protocols 15-224
15.10.1 IOR 15-224
15.10.2 Mechanism Tags 15-225
15.10.3 Association Options 15-225
15.10.4 Cryptographic Profiles 15-226
15.10.4.1 Key Establishment Algorithms 15-227
15.10.4.2 Common Message Protection Algorithms 15-227
15.10.4.3 Cryptographic Profiles Supported by CSI Protocols 15-227
15.10.5 Security Name 15-227
15.10.6 Security Administration Domains. 15-228
15.10.7 Mapping of Common Elements to the SECIOP Protocol 15-228
15.10.7.1 Basic Token Format 15-228
15.10.7.2 Inner Context Tokens 15-229
15.10.8 CSI Protocols 15-230
15.10.8.1 SPKM Protocol 15-230
15.10.8.2 GSS Kerberos Protocol 15-231
15.10.8.3 CSI-ECMA Protocol 15-231
15.11 SPKM Protocol 15-231
15.11.1 Cryptographic Profiles 15-231
15.11.1.1 MD5_RSA 15-232
15.11.1.2 MD5_DES_CBC 15-232
15.11.1.3 DES_CBC 15-232
15.11.1.4 MD5_DES_CBC_SOURCE 15-232
15.11.1.5 DES_CBC_SOURCE 15-232
15.11.2 IOR Encoding 15-232
15.11.3 Using SPKM for SECIOP. 15-233
15.11.3.1 The Initial Context Token 15-233
15.11.3.2 The Final Context Token 15-233
15.11.3.3 The Continuation Context Token 15-234
15.11.3.4 The Message Protection Token 15-234
15.11.3.5 The Context Delete Token 15-234
15.12 GSS Kerberos Protocol. 15-234
15.12.1 Cryptographic Profiles 15-234
15.12.1.1 DES_CBC_DES_MAC 15-234
15.12.1.2 DES_CBC_MD5 15-234
15.12.1.3 DES_MAC 15-234
15.12.1.4 MD5 15-235
15.12.2 Mandatory and Optional Cryptographic Profiles 15-235
15.12.3 IOR Encoding 15-235

Contents

15.12.4 SECIOP Tokens 15-235
15.12.4.1 The Initial Context Token 15-236
15.12.4.2 The Final Context Token 15-236
15.12.4.3 The Continuation Context Token 15-236
15.12.4.4 The Message Protection Token 15-236
15.13 CSI-ECMA Protocol 15-237
15.13.1 Concepts 15-237
15.13.1.1 Separation of Concerns 15-237
15.13.2 Security Attributes 15-238
15.13.2.1 Privilege Attributes 15-238
15.13.2.2 Miscellaneous Attributes 15-238
15.13.3 Target Access Enforcement Function 15-239
15.13.4 Basic and Dialogue Keys 15-239
15.13.5 Key Distribution Schemes 15-239
15.13.5.1 Basic Symmetric Key Distribution Scheme 15-240
15.13.5.2 Symmetric Key Distribution with Asymmetric KDS 15-240
15.13.5.3 Full Public Key Scheme 15-240
15.13.6 Cryptographic Algorithms and Profiles 15-240
15.13.7 PAC Protection and Delegation - Outline . . . 15-242
15.13.8 PPID Method 15-243
15.13.9 PV/CV Delegation Method. 15-243
15.13.9.1 Restrictions 15-244
15.13.10 Mechanism Identifiers and IOR Encoding . 15-244
15.13.11 Security Names 15-244
15.13.11.1 Kerberos Naming 15-245
15.13.11.2 Directory Naming 15-245
15.13.12 SECIOP Tokens When Using CSI-ECMA . 15-245
15.13.13 Initial Context Token. 15-246
15.13.13.1 TargetAEF Part 15-248
15.13.14 TargetResultToken. 15-249
15.13.15 ErrorToken 15-250
15.13.16 Per-Message Tokens 15-250
15.13.16.1 MICToken 15-252
15.13.16.2 WrapToken 15-252
15.13.17 ContextDeleteToken 15-252
15.13.18 Security Attributes 15-253
15.13.18.1 Data Structures 15-253
15.13.18.2 Attribute Types 15-254
15.13.19 Privilege and Miscellaneous Attribute Definitions. 15-255
15.13.20 Qualifier Attributes 15-255
15.13.21 Target Names 15-256
15.13.22 PAC Format. 15-256
15.13.23 Common Contents Fields 15-257

15.13.24 Specific Certificate Contents for PACs 15-257
15.13.24.1 Protection Methods 15-259
15.13.24.2 External Control Values Construct . . 15-260
15.13.25 Check Value 15-261
15.13.26 Basic Key Distribution 15-262
15.13.27 Keying Information Syntax 15-262
15.13.28 Summary of Key Distribution Schemes . . . 15-263
15.13.29 CSI-ECMA Secret Key Mechanism 15-264
15.13.29.1 Profile of Ticket as Used in SymmIntradomain Scheme 15-264
15.13.30 CSI-ECMA Hybrid Mechanism 15-265
15.13.30.1 Hybrid Inter-domain Key Distribution Scheme Data Elements 15-266
15.13.30.2 Key Establishment Data Elements . . 15-267
15.13.30.3 Key Establishment Algorithm 15-268
15.13.30.4 Profile of Ticket as Used in Hybrid Interdomain Scheme 15-268
15.13.31 CSI-ECMA Public Mechanism 15-269
15.13.31.1 Profile of SPKM_REQ Used in Public Key Mechanism 15-270
15.13.32 Dialogue Key Block 15-271
15.14 Integrating SSL with CORBA Security 15-272
15.14.1 Introduction 15-272
15.14.2 Cryptographic Profiles 15-272
15.14.3 IOR Encoding 15-272
15.14.4 Relation to SECIOP 15-273
15.14.5 DCE-CIOP with Security 15-273
15.14.6 Goals of Secure DCE-CIOP 15-273
15.14.7 Secure DCE-CIOP Overview 15-273
15.14.7.1 IOR Security Components for DCE-CIOP 15-274
15.14.7.2 TAG_ASSOCIATION_OPTIONS . . . 15-276
15.14.7.3 TAG_SEC_NAME 15-278
15.14.8 DCE RPC Security Services 15-278
15.14.8.1 Secure DCE-CIOP Operational Semantics 15-280

16. Trading Object Service Specification 16-1
16.1 Overview 16-2
16.1.1 Diversity and Scalability 16-3
16.1.2 Linking Traders 16-3
16.1.3 Policy 16-3
16.1.4 Additional ObjectID 16-4
16.2 Concepts and Data Types 16-4
16.2.1 Exporter 16-4

Contents

16.2.2 Importer 16-4
16.2.3 Service Types 16-4
16.2.3.1 Service Type Model 16-5
16.2.4 Properties 16-7
16.2.5 Service Offers. 16-7
16.2.5.1 Modifiable Properties 16-8
16.2.5.2 Dynamic Properties 16-8
16.2.6 Offer Identifier 16-9
16.2.7 Offer Selection 16-9
16.2.7.1 Standard Constraint Language 16-9
16.2.7.2 Preferences 16-10
16.2.7.3 Links 16-11
16.2.7.4 Policies 16-12
16.2.7.5 Trader Policies 16-16
16.2.7.6 Link Follow Behavior 16-16
16.2.7.7 Importer Policies 16-17
16.2.7.8 Exporter Policies 16-18
16.2.7.9 Link Creation Policies 16-18
16.2.8 Interworking Mechanisms 16-18
16.2.8.1 Link Traversal Control 16-18
16.2.8.2 Federated Query Example 16-19
16.2.8.3 Proxy Offers 16-20
16.2.9 Trader Attributes. 16-21
16.3 Exceptions 16-23
16.3.1 For CosTrading module 16-23
16.3.1.1 Exceptions used in more than one interface 16-23
16.3.1.2 Additional Exceptions for Lookup Interface 16-24
16.3.1.3 Additional Exceptions For Register Interface 16-25
16.3.1.4 Additional Exceptions for Link Interface 16-26
16.3.1.5 Additional Exceptions for Proxy Offer Interface 16-27
16.3.2 For CosTradingDynamic module 16-27
16.3.3 For CosTradingRepos module 16-27
16.4 Abstract Interfaces 16-28
16.4.1 TraderComponents 16-28
16.4.2 SupportAttributes 16-29
16.4.3 ImportAttributes 16-29
16.4.4 LinkAttributes. 16-30
16.5 Functional Interfaces 16-30
16.5.1 Lookup 16-30
16.5.1.1 Query Operation 16-31
16.5.2 Offer Iterator. 16-35
16.5.2.1 Signature 16-35
16.5.2.2 Function 16-36

16.5.3 Register 16-36
16.5.3.1 Export Operation 16-39
16.5.3.2 Withdraw Operation 16-41
16.5.3.3 Describe Operation 16-41
16.5.3.4 Modify Operation 16-42
16.5.3.5 Withdraw Using Constraint Operation . 16-44
16.5.3.6 Resolve Operation 16-45
16.5.4 Offer Id Iterator 16-45
16.5.4.1 Signature 16-45
16.5.4.2 Function 16-46
16.5.5 Admin 16-46
16.5.5.1 Attributes and Set Operations 16-48
16.5.5.2 List Offers Operation 16-48
16.5.5.3 List Proxies Operation 16-48
16.5.6 Link 16-49
16.5.6.1 Add_Link Operation 16-51
16.5.6.2 Remove Link Operation 16-52
16.5.6.3 Describe Link Operation 16-52
16.5.6.4 List Links Operation 16-53
16.5.6.5 Modify Link Operation 16-53
16.5.7 Proxy 16-54
16.5.7.1 Export Proxy Operation 16-55
16.5.7.2 Withdraw Proxy Operation 16-58
16.5.7.3 Describe Proxy Operation 16-58
16.6 Service Type Repository 16-59
16.7 Dynamic Property Evaluation interface 16-67
16.8 Conformance Criteria 16-68
16.8.1 Conformance Requirements for Trading Interfaces as Server 16-69
16.8.1.1 Lookup Interface 16-69
16.8.1.2 Register Interface 16-69
16.8.1.3 Admin Interface 16-70
16.8.1.4 Link Interface 16-70
16.8.1.5 Proxy Interface 16-70
16.8.2 Conformance Requirements for Implementation Conformance Classes 16-71
16.8.2.1 Query Trader 16-71
16.8.2.2 Simple Trader 16-72
16.8.2.3 Stand-alone Trader 16-72
16.8.2.4 Linked Trader 16-72
16.8.2.5 Proxy Trader 16-73
16.8.2.6 Full-service Trader 16-73

17. Object Collection Specification 17-1

17.1 Overview 17-2
17.2 Service Structure 17-2
17.2.1 Combined Property Collections 17-3
17.2.1.1 Restricted Access Collections 17-4
17.2.1.2 Collection Factories 17-5

Contents

17.2.2 Iterators. 17-5
17.2.3 Function Interfaces 17-7
17.2.3.1 Collectible Elements and Type Safety . 17-7
17.2.3.2 Collectible Elements and the Operations Interface 17-7
17.2.3.3 Collectible Elements of Key Collections 17-8
17.2.4 List of Interfaces Defined 17-8

17.3 Combined Collections 17-10
17.3.1 Combined Collections Usage Samples 17-10
17.3.1.1 Bag, SortedBag 17-10
17.3.1.2 EqualitySequence 17-11
17.3.1.3 Heap 17-11
17.3.1.4 KeyBag, KeySortedBag 17-11
17.3.1.5 KeySet, KeySortedSet 17-12
17.3.1.6 Map, SortedMap 17-12
17.3.1.7 Relation, SortedRelation 17-13
17.3.1.8 Set, SortedSet 17-13
17.3.1.9 Sequence 17-13

17.4 Restricted Access Collections 17-14
17.4.1 Restricted Access Collections Usage Samples . 17-14
17.4.1.1 Deque 17-14
17.4.1.2 PriorityQueue 17-14
17.4.1.3 Queue 17-15
17.4.1.4 Stack 17-15

17.5 The CosCollection Module. 17-15
17.5.1 Interface Hierarchies. 17-15
17.5.1.1 Collection Interface Hierarchies 17-15
17.5.1.2 Iterator Hierarchy 17-18
17.5.2 Exceptions and Type Definitions 17-19
17.5.3 Abstract Collection Interfaces 17-21
17.5.3.1 The Collection Interface 17-21
17.5.3.2 The OrderedCollection Interface 17-28
17.5.3.3 The SequentialCollection Interface 17-31
17.5.3.4 The SortedCollection Interface 17-37
17.5.3.5 The EqualityCollection Interface 17-37
17.5.3.6 The KeyCollection Interface 17-42
17.5.3.7 The EqualityKeyCollection Interface . . 17-50
17.5.3.8 The KeySortedCollection Interface 17-50
17.5.3.9 The EqualitySortedCollection Interface 17-52
17.5.3.10 The EqualityKeySortedCollection Interface 17-55
17.5.3.11 The EqualitySequentialCollection Interface 17-55
17.5.4 Concrete Collections Interfaces 17-56
17.5.4.1 The KeySet Interface 17-57
17.5.4.2 The KeyBag Interface 17-57
17.5.4.3 The Map Interface 17-57
17.5.4.4 The Relation Interface 17-60
17.5.4.5 The Set Interface 17-61
17.5.4.6 The Bag Interface 17-61

17.5.4.7 The KeySortedSet Interface 17-61
17.5.4.8 The KeySortedBag Interface 17-62
17.5.4.9 The SortedMap Interface 17-62
17.5.4.10 The SortedRelation Interface 17-62
17.5.4.11 The SortedSet Interface 17-63
17.5.4.12 The SortedBag Interface 17-63
17.5.4.13 The Sequence Interface 17-63
17.5.4.14 The EqualitySequence Interface 17-63
17.5.4.15 The Heap Interface 17-64
17.5.5 Restricted Access Collection Interfaces 17-64
17.5.6 Abstract RestrictedAccessCollection Interface . 17-64
17.5.6.1 The RestrictedAccessCollection Interface 17-64
17.5.7 Concrete Restricted Access Collection Interfaces 17-65
17.5.7.1 The Queue Interface 17-65
17.5.7.2 The Dequeue Interface 17-66
17.5.7.3 The Stack Interface. 17-67
17.5.7.4 The PriorityQueue Interface 17-68
17.5.8 Collection Factory Interfaces 17-69
17.5.8.1 The CollectionFactory and Collection Factories Interfaces 17-70
17.5.8.2 The RACollectionFactory and RA CollectionFactories Interfaces 17-74
17.5.8.3 The KeySetFactory Interface 17-74
17.5.8.4 The KeyBagFactory Interface 17-75
17.5.8.5 The MapFactory Interface 17-75
17.5.8.6 The RelationFactory Interface 17-76
17.5.8.7 The SetFactory Interface 17-76
17.5.8.8 The BagFactory Interface 17-77
17.5.8.9 The KeySortedSetFactory Interface . . . 17-77
17.5.8.10 The KeySortedBagFactory Interface . . 17-78
17.5.8.11 The SortedMapFactory Interface 17-78
17.5.8.12 The SortedRelationFactory Interface . 17-79
17.5.8.13 The SortedSetFactory Interface 17-79
17.5.8.14 The SortedBagFactory Interface 17-80
17.5.8.15 The SequenceFactory Interface 17-80
17.5.8.16 The EqualitySequence Factory Interface 17-81
17.5.8.17 The HeapFactory Interface 17-81
17.5.8.18 The QueueFactory Interface 17-82
17.5.8.19 The StackFactory Interface 17-82
17.5.8.20 The DequeFactory Interface 17-82
17.5.8.21 The PriorityQueueFactory Interface . . 17-82
17.5.9 Iterator Interfaces 17-83
17.5.9.1 Iterators as pointer abstraction 17-83
17.5.9.2 Iterators and support for generic programming 17-84
17.5.9.3 Iterators and performance 17-84
17.5.9.4 The Managed Iterator Model 17-84
17.5.9.5 The Iterator Interface 17-85
17.5.9.6 The OrderedIterator Interface 17-96
17.5.9.7 The SequentialIterator Interface 17-104
17.5.9.8 The KeyIterator Interface 17-106
17.5.9.9 The EqualityIterator Interface 17-109

Contents

17.5.9.10 The EqualityKeyIterator Interface 17-110
17.5.9.11 The SortedIterator Interface 17-110
17.5.9.12 The KeySortedIterator Interface 17-110
17.5.9.13 The EqualitySortedIterator Interface . . 17-113
17.5.9.14 The EqualityKeySortedIterator Interface 17-115
17.5.9.15 The EqualitySequentialIterator Interface 17-115
17.5.10 Function Interfaces 17-117
17.5.10.1 The Operations Interface 17-117
17.5.10.2 The Command and Comparator Interface 17-121
17.5.10.3 Identification and Justification of Differences 17-122
17.5.10.4 CosQueryCollection Module Detailed Comparison 17-124
17.5.10.5 Containers 17-131
17.5.10.6 Algorithms 17-132
17.5.10.7 Iterators 17-132
17.5.10.8 Consideration on choice 17-133

List of Figures

Figure 2-1	An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.	2-3
Figure 3-1	A Naming Graph	3-2
Figure 3-2	The CosNaming Module	3-6
Figure 3-3	The Names Library Interface in PIDL	3-14
Figure 4-1	Push-style Communication Between a Supplier and a Consumer	4-7
Figure 4-2	Pull-style Communication Between a Supplier and a Consumer	4-7
Figure 4-3	The OMG IDL Module CosEventComm	4-8
Figure 4-4	Push-style Communication Between a Supplier and an Event Channel, and a Consumer and an Event Channel	4-11
Figure 4-5	Pull-style communication between a supplier and an event channel and a consumer and the event channel	4-11
Figure 4-6	Push-style Communication Between a Supplier and an Event Channel, and Pull-style Communication Between a Consumer and an Event Channel	4-12
Figure 4-7	An Event Channel with Multiple Suppliers and Multiple Consumers	4-12
Figure 4-8	A newly created event channel. The channel has no suppliers or consumers	4-13
Figure 4-9	State diagram of a proxy	4-14

Figure 4-10	The CosEventChannelAdmin Module	4-16
Figure 4-11	Typed Push-style Communication Between a Supplier and a Consumer	4-20
Figure 4-12	Typed Pull-style Communication Between a Supplier and a Consumer	4-21
Figure 4-13	The IDL Module CosTypedEventComm	4-22
Figure 4-14	The CosTypedEventChannelAdmin Module	4-25
Figure 5-1	Roles in the Persistent Object Service	5-1
Figure 5-2	Major Components of the POS and their Interactions	5-8
Figure 5-3	The CosPersistencePID Module	5-9
Figure 5-4	TheCosPersistencePO Module	5-12
Figure 5-5	The CosPersistencePOM Module	5-15
Figure 5-6	Example to illustrate POMFunctions	5-18
Figure 5-7	The CosPersistencePDS Module	5-20
Figure 5-8	Direct Access Protocol Interfaces	5-21
Figure 5-9	The CosPersistencePDS_DA Module	5-22
Figure 5-10	Structure of a DDO	5-31
Figure 5-11	The CosPersistenceDDO Module	5-32
Figure 5-12	The CosPersistenceDS_CLI Module	5-35
Figure 6-1	Life Cycle service defines how a client can create an object “over there”.	6-1
Figure 6-2	Life Cycle Service defines how a client can move or copy an object over there.	6-2
Figure 6-3	The object life cycle problem for graphs of objects is to determine the boundaries of a graph of objects and operate on that graph. In the above example, a document contains a graphic and a logo, refers to a dictionary and is contained in a folder.	6-3
Figure 6-4	To create an object “over there” a client must possess an object reference to a factory over there. The client simply issues a request on the factory.	6-4
Figure 6-5	An example of a document factory interface. This interface is defined for clients as a part of application development.	6-5
Figure 6-6	To delete an object, a client must posses an object reference supporting the *LifeCycleObject* interface and issues a remove request on the object.	6-6

Figure 6-7	Life cycle services define how a client can move or copy an object from here to there.	6-7
Figure 6-8	The *FactoryFinder* interface can be “mixed in” with interfaces of more powerful finding services.	6-8
Figure 6-9	The CosLifeCycle Module	6-10
Figure 6-10	The Life Cycle service provides a generic creation capability.	6-15
Figure 6-11	Factories assemble resources for the execution of an object.	6-19
Figure 6-12	In an administered environment, factory implementations can delegate the creation problem to a generic factory.	6-20
Figure 6-13	The copy and move operations are passed a FactoryFinder to represent "there."	6-21
Figure 8-1	Externalization control flow when streamable object is not in a graph of related objects	8-5
Figure 8-2	Externalization control flow when streamable object is a node in a graph of related objects	8-6
Figure 8-3	Internalization control flow when object is not in a graph of related objects	8-7
Figure 8-4	Internalization control flow when object is in a graph of related objects	8-8
Figure 8-5	Object Externalization Service Booch Class (=Interface) Diagram	8-10
Figure 8-6	Internalizing a node returns the new object and the corresponding roles.	8-24
Figure 9-1	Base relationships	9-7
Figure 9-2	Navigation functionality of base relationships	9-8
Figure 9-3	An example graph of related objects	9-9
Figure 9-4	Relationship interface hierarchy	9-10
Figure 9-5	Role interface hierarchy	9-10
Figure 9-6	Simple relationship type: documents reference books	9-14
Figure 9-7	Simple relationship instance: my document references the book “War and Peace“	9-14
Figure 9-8	A ternary check-out relationship type between books, libraries and persons	9-15

Figure 9-9 An unsatisfactory representation of the ternary check-out relationship using binary relationships 9-16

Figure 9-10 Another unsatisfactory representation 9-16

Figure 9-11 Creating a role for an object 9-17

Figure 9-12 A fully established binary relationship 9-17

Figure 9-13 The CosObjectIdentity Module 9-19

Figure 9-14 The CosRelationships Module 9-21

Figure 9-15 Two binary one-to-many containment relationships. 9-23

Figure 9-16 An example graph of related objects 9-34

Figure 9-17 A traversal of a graph for compound copy operation. . . . 9-37

Figure 9-18 How deep, shallow and none propagation values affect nodes, roles and relationships. 9-38

Figure 9-19 The CosGraphs Module 9-39

Figure 9-20 The CosContainment Module 9-48

Figure 9-21 The CosReference Module 9-50

Figure 10-1 Application Including Basic Elements 10-4

Figure 10-2 This figure illustrates the major components and interfaces of the Transaction Service 10-12

Figure 10-3 X/Open client 10-45

Figure 10-4 X/Open server 10-45

Figure 10-5 Example 10-46

Figure 10-6 Model interoperability example 10-68

Figure 11-1 Query Evaluators: Nesting and Federation 11-3

Figure 11-2 Queryable Collections 11-5

Figure 11-3 SQL Query = OQL 11-8

Figure 11-4 Collection interface structure 11-10

Figure 11-5 Query Framework interface hierarchy/structure 11-11

Figure 11-6 CosQueryCollection Module 11-14

Figure 11-7 Query Evaluator and Queryable Collection 11-20

Figure 11-8 Query Manager and Query Object 11-21

Figure 11-9 QueryLanguageType Interface Hierarchy 11-24

Figure 12-1 Licensing Service Relationships 12-7

Figure 12-2 Licensing Service Instance Diagram 12-14

Figure 12-3	Licensing Event Trace Diagram	12-16
Figure 12-4	CosLicensingManager Module	12-17
Figure 13-1	Data types	13-5
Figure 13-2	PropertySet interface exceptions	13-7
Figure 13-3	Operations used to define new properties or set new values.	13-9
Figure 13-4	Operations used to retrieve property names and values	13-11
Figure 13-5	Operations used to delete properties.	13-12
Figure 13-6	is_property_defined operation.	13-14
Figure 13-7	Operations used to retrieve information related to constraints	13-15
Figure 13-8	Operations used to define new properties or values	13-16
Figure 13-9	Operations used to get and set property mode	13-18
Figure 13-10	reset operation.	13-19
Figure 13-11	next_one and next_n operations (properties)	13-20
Figure 13-12	destroy operation	13-20
Figure 13-13	reset operation.	13-20
Figure 13-14	next_one, next_n operations (PropertyNames)	13-21
Figure 13-15	destroy operation	13-21
Figure 13-16	PropetySetFactory interface	13-21
Figure 13-17	PropertySetDefFactory interface.	13-22
Figure 14-1	General Object Model for Service	14-3
Figure 14-2	Object Model for Time Service	14-5
Figure 14-3	Illustration of Interval Overlap	14-8
Figure 14-4	Object Model of Timer Event Service	14-13
Figure 14-5	Time Service and Proxies	14-19
Figure 15-1	Structure of the Specification	15-10
Figure 15-2	Modules and Their Relation to Layers of the Architecture	15-18
Figure 15-3	A Security model for object systems	15-19
Figure 15-4	Credential containing security attributes	15-21
Figure 15-5	Invocation of Target Object via ORB	15-21
Figure 15-6	Message protection	15-23

Figure 15-7 Access control model 15-25
Figure 15-8 Authorization model 15-26
Figure 15-9 Auditing model 15-29
Figure 15-10 Delegation model 15-30
Figure 15-11 No delegation 15-33
Figure 15-12 Simple delegation 15-33
Figure 15-13 Composite delegation 15-33
Figure 15-14 Combined privileges delegation 15-34
Figure 15-15 Traced delegation 15-34
Figure 15-16 Proof of receipt 15-36
Figure 15-17 Non-repudiation services 15-37
Figure 15-18 Security policy domains 15-38
Figure 15-19 Policy domain hierarchies 15-39
Figure 15-20 Federated policy domains 15-40
Figure 15-21 System- and application-enforced policies 15-40
Figure 15-22 Overlapping policy domains 15-40
Figure 15-23 Framework of domains 15-43
Figure 15-24 Structural model 15-50
Figure 15-25 ORB services 15-51
Figure 15-26 Object reference 15-53
Figure 15-27 Domain objects 15-54
Figure 15-28 Controlled relationship 15-57
Figure 15-29 Object encapsulation 15-57
Figure 15-30 Authentication 15-59
Figure 15-31 Multiple credentials 15-61
Figure 15-32 Changing security attributes 15-62
Figure 15-33 Making a secure invocation 15-63
Figure 15-34 Target object security 15-64
Figure 15-35 Security-unaware intermediate object 15-65
Figure 15-36 Security-aware intermediate object 15-66
Figure 15-37 access_allowed application 15-68
Figure 15-38 get_policy application 15-68

Figure 15-39	audit_write application	15-69
Figure 15-40	Audit decision object	15-69
Figure 15-41	set_NR_features operation	15-70
Figure 15-42	generate_token operation	15-71
Figure 15-43	Non-repudiation service	15-73
Figure 15-44	verify_evidence operation	15-74
Figure 15-45	Proof of origin message	15-74
Figure 15-46	Managing security policies	15-78
Figure 15-47	Securing invocations	15-80
Figure 15-48	get_policy operation	15-81
Figure 15-49	ORB Security Services	15-82
Figure 15-50	Access decision object	15-83
Figure 15-51	Target objects sharing security names	15-85
Figure 15-52	Object created by application or factory	15-86
Figure 15-53	Relationship between main objects	15-87
Figure 15-54	Security Functionality Implemented by Security Service Objects	15-157
Figure 15-55	Model for Secure Interoperability	15-173
Figure 15-56	Security Functionality Implemented by Security Service Objects	15-196
Figure 15-57	Attribute Mapper Diagram	15-198
Figure 15-58	Position of SECIOP Protocol	15-204
Figure 15-59	Sublayers of SECIOP	15-204
Figure 15-60	Architectural Assumptions	15-205
Figure 15-61	Initial Context Token	15-246
Figure 15-62	Generalized Certificate’s Structural Components	15-256
Figure 15-63	Normal System Interactions	15-336
Figure 15-64	Distributed TCB	15-338
Figure 15-65	Base Protection and Communications	15-340
Figure 15-66	Protection Boundaries	15-341
Figure 15-67	Distribution of Security Functionality and Trust	15-349
Figure 15-68	Intended Use by AccessDecision	15-363
Figure 15-69	Supporting Overlapping Access Policy Domains	15-364

Figure 15-70	Hierarchical Domains	15-365
Figure 15-71	Retrieving Granted Rights	15-366
Figure 15-72	Mutual Authentication	15-370
Figure 15-73	Confidential Message with Context Establishment	15-371
Figure 15-74	Fragmented GIOP Request with Context Establishment	15-372
Figure 16-1	Interactions between a trader and its clients	16-1
Figure 16-2	Property Strength	16-5
Figure 16-3	Pipeline View of Trader Query Steps and Cardinality Constraint Application	16-15
Figure 16-4	Flow of a query through a trader graph	16-19
Figure 17-1	Collections Interfaces Hierarchy	17-17
Figure 17-2	Restricted Access Collections Interface Hierarchy	17-17
Figure 17-3	Iterator Interface Hierarchy	17-18
Figure 17-4	Inheritance Relationships	17-126

List of Tables

Table 3-5 Exceptions Raised by Binding Operations 3-9

Table 3-6 Exceptions Raised by Resolve Operation. 3-10

Table 3-7 Exceptions Raised by Unbind Operation 3-10

Table 3-8 Exceptions Raised by Creating New Contexts 3-11

Table 6-1 Suggested Conventions for Factory Finder Keys 6-14

Table 6-2 Suggested Conventions for Generic Factory Keys 6-16

Table 6-3 Suggested Criteria. 6-17

Table 8-1 Client Functional Interfaces support client's model of externalization 8-11

Table 8-2 Service Construction Interfaces support service implementation's model of externalization. 8-11

Table 8-3 Compound Externalization Interfaces support service implementation's model of graph externalization . 8-12

Table 8-4 CORBA Tag Byte Values and Data Formats 8-30

Table 9-1 Interfaces Defined in the CosObjectIdentity Module 9-11

Table 9-2 Interfaces Defined in the CosRelationships Module 9-11

Table 9-3 Interfaces Defined in the CosGraphs Module. 9-12

Table 9-4 Interfaces Defined in the CosContainment Module 9-12

Table 9-5 Interfaces Defined in the CosReference Module 9-13

Table 10-1 Use of Transaction Service Functionality. 10-36

Table 11-1 Interfaces Defined in the CosQueryCollection Module 11-12

Table 12-1 Exceptions Raised by Licensing Service Operations. 12-19

Table 13-1 Property Service Interfaces. 13-3

Table 13-2 Exceptions Raised by Define Operations 13-10

Table 13-3 Exceptions Raised by List and Get Properties Operations 13-12

Table 13-4 Exceptions Raised by delete_properties Operations. 13-13

Table 13-5 Exceptions Raised by define Operations 13-17

Table 13-6 Exceptions Raised by Get and Set Mode Operations. 13-19

Table 15-1 Feature Packages and Modules. 15-17

Table 15-2 DomainAccessPolicy 15-138

Table 15-3 User Privilege Attributes (Not Defined by This Specification) 15-138

Table 15-4 DomainAccessPolicy (with Privilege Attributes). 15-139

Table 15-5 DomainAccessPolicy (with Delegate Entry) 15-139

Table 15-6 Interface Instances 15-140

Table 15-7 DomainAccessPolicy (with Required Rights Mapping). 15-140

Table 15-8 RequiredRights for Interfaces c1, c2 and c3 15-140

Table 15-9 Standard Audit Policy. 15-145

Table 15-10 Definition of Association Options 15-180

Table 15-11 IOR Example 15-181

Table 15-12 CSI Functionality and Protocols. 15-185

Table 15-13	SECIOP Context Management Finite State Machine (Table 1)	15-218
Table 15-14	SECIOP Context Management Finite State Machine (Table 2)	15-221
Table 15-15	Summary of Algorithm Usage	15-241
Table 15-16	Summary of Algorithm Classes	15-241
Table 15-17	Cryptographic Algorithm Usage Profiles	15-242
Table 15-18	Privilege and Miscellaneous Attributes	15-255
Table 15-19	Syntaxes Used for targetKDSpart and targetPart	15-264
Table 15-20	Kerberos Ticket's Mechanism Fields	15-264
Table 15-21	Ticket as Used in Hybrid Interdomain Scheme	15-268
Table 15-22	SPKM-REQ Used in Public Key Mechanism	15-270
Table 15-23	Association Option Mapping to DCE Security	15-276
Table 15-24	Relationship between DCE-CIOP and DCE RPC Authorization Service Identifiers	15-279
Table 15-25	Attribute Values	15-308
Table 15-26	CORBA Rights Family Values	15-308
Table 15-27	System Audit Events	15-309
Table 15-28	Mechanism Ids	15-310
Table 15-29	Cryptographic Profile Ids	15-310
Table 15-30	CORBA Security Functionality Checklist	15-318
Table 15-31	CORBA Secure Interoperability Checklist	15-318
Table 15-32	CORBA Security Functionality Checklist	15-354
Table 15-33	CORBA Secure Interoperability Checklist	15-354
Table 16-1	Preferences	16-10
Table 16-2	Scoping Policies	16-13
Table 16-3	Capability Supported Policies	16-15
Table 16-4	Trader Attributes	16-21
Table 16-5	Primary/Secondary Policy Parameters	16-56
Table 17-1	Interfaces derived from combinations of collection properties	17-4
Table 17-2	Iterators and Collections	17-19

Table 17-3	Collection interfaces and the iterator interfaces supported	17-27
Table 17-4	Implementation Category Examples	17-72
Table 17-5	Required element and key-type specific user-defined information for KeySetFactory. []- implied by key_compare.	17-75
Table 17-6	Required element and key-type specific user-defined information for KeyBagFactory. []- implied by key_compare.	17-76
Table 17-7	Required element and key-type specific user-defined information for MapFactory. []- implied by key_compare.	17-76
Table 17-8	Required element and key-type specific user-defined information for RelationFactory.[]- implied by key_compare.	17-77
Table 17-9	Required element and key-type specific user-defined information for SetFactory.[]- implied by compare.	17-77
Table 17-10	Required element and key-type specific user-defined information for BagFactory.[]- implied by compare.	17-78
Table 17-11	Required element and key-type specific user-defined information for KeySortedSetFactory.[]- implied by key_compare.	17-78
Table 17-12	Required element and key-type specific user-defined information for KeySortedBagFactory.[]- implied by key_compare.	17-79
Table 17-13	Required element and key-type specific user-defined information for SortedMapFactory.[]- implied by key_compare.	17-79
Table 17-14	Required element and key-type specific user-defined information for SortedRelationFactory.[]- implied by key_compare.	17-80
Table 17-15	Required element and key-type specific user-defined information for SortedSetFactory. []- implied by compare.	17-80
Table 17-16	Required element and key-type specific user-defined information for SortedBagFactory. []- implied by compare.	17-81
Table 17-17	Required element and key-type specific user-defined information for EqualitySequenceFactory.	17-82

Table 17-18 Required element and key-type specific user-defined information for PriorityQueueFactory. [] - implied by key_compare. 17-83

Preface

0.1 About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

0.1.1 Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 750 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

0.1.2 X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

0.2 Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, “Need for Object Services.”

0.3 Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG’s vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in this manual.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities is contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects,** which are products of a single vendor on in-house development group which controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

0.3.1 What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

0.4 Associated Documents

The CORBA documentation set includes the following books:

- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the object services.
- *CORBAfacilities: Common Facilities Architecture* contains information about the design of Common Facilites; it provides the framework for Common Facility specifications.
- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

To obtain books in the documentation set, or other OMG publications, refer to the enclosed subscription card or contact the Object Management Group, Inc. at:

OMG Headquarters
492 Old Connecticut Path
Framingham, MA 01701
USA
Tel: +1-508-820-4300
pubs@omg.org
http://www.omg.org

0.5 Structure of this Manual

In addition to this preface, *CORBAservices: Common Object Services* contains the following chapters:

Overview provides an introduction to the CORBA object services, including a summary of features for each service.

General Design Principles provides information about the principles that were used in designing each service; explains the dependencies among services; and explains how Object Services relate to each other, CORBA, and industry standards in general.

Chapters 3 through 16 each contain a specification for the following Object Services:

- Naming
- Event
- Persistent Object
- Life Cycle
- Concurrency Control
- Externalization
- Relationship
- Transaction
- Query
- Licensing
- Property
- Time
- Security
- Trading
- Collections

0.6 Acknowledgements

The following companies submitted parts of the specifications that were approved by the Object Management Group to become *CORBAservices:*

AT&T/Lucent Technologies
AT&T/NCR
BNR Europe Limited
Cooperative Research Centre for Distributed Systems Technology (DTSC Pty Ltd.)
Digital Equipment Corporation
Expersoft Corporation
Gradient Technologies, Inc.
Groupe Bull
Hewlett-Packard Company
HyperDesk Corporation
ICL PLC
Ing. C. Olivetti & C.Sp

International Business Machines Corporation
International Computers Limited
Iona Technologies Ltd.
Itasca Systems, Inc.
Nortel Limited
Novell, Inc.
O2 Technologies, SA
Object Design, Inc.
Objectivity, Inc.
Odyssey Research Associates, Inc.
Ontos, Inc.
Oracle Corporation
Persistence Software, Inc.
Servio Corporation
Siemens Nixdorf Informationssysteme AG
Sun Microsystems, Inc.
SunSoft, Inc.
Sybase, Inc.
Taligent, Inc.
Tandem Computers, Inc.
Teknekron Software Systems, Inc.
Tivoli Systems, Inc.
Transarc Corporation
Versant Object Technology Corporation

Overview 1

1.1 Summary of Key Features

1.1.1 Naming Service

- The Naming Service provides the ability to bind a name to an object relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. To resolve a name is to determine the object associated with the name in a given context.
- Through the use of a very general model and dealing with names in their structural form, naming service implementations can be application specific or be based on a variety of naming systems currently available on system platforms.
- Graphs of naming contexts can be supported in a distributed, federated fashion. The scalable design allows the distributed, heterogeneous implementation and administration of names and name contexts.
- Because name component attribute values are not assigned or interpreted by the naming service, higher levels of software are not constrained in terms of policies about the use and management of attribute values.
- Through the use of a "names library," name manipulation is simplified and names can be made representation-independent thus allowing their representation to evolve without requiring client changes.
- Application localization is facilitated by name syntax-independence and the provision of a name "kind" attribute.

1.1.2 Event Service

- The Event Service provides basic capabilities that can be configured together in a very flexible and powerful manner. Asynchronous events (decoupled event suppliers and consumers), event "fan-in," notification "fan-out," and (through appropriate event channel implementations) reliable event delivery are supported.
- The Event Service design is scalable and is suitable for distributed environments. There is no requirement for a centralized server or dependency on any global service.
- The Event Service interfaces allow implementations that provide different qualities of service to satisfy different application requirements. In addition, the event service does not impose higher level policies (e.g., specific event types) allowing great flexibility on how it is used in a given application environment.
- Both push and pull event delivery models are supported: that is, consumers can either request events or be notified of events, whichever is needed to satisfy application requirements. There can be multiple consumers and multiple suppliers events.
- Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers.
- The event channel interface can be subtyped to support extended capabilities. The event consumer-supplier interfaces are symmetric, allowing the chaining of event channels (for example, to support various event filtering models). Event channels can be chained by third-parties.
- Typed event channels extend basic event channels to support typed interaction.
- Because event suppliers, consumers and channels are objects, advantage can be taken of performance optimizations provided by ORB implementations for local and remote objects. No extension is required to CORBA.

1.1.3 Life Cycle Service

- The Life Cycle Service defines conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, life cycle services define services and conventions that allow clients to perform life cycle operations on objects in different locations.
- The client's model of creation is defined in terms of factory objects. A factory is an object that creates another object. Factories are *not* special objects. As with any object, factories have well-defined OMG IDL interfaces and implementations in some programming language.
- The Life Cycle Service defines an interface for a generic factory. This allows for the definition of standard creation services.
- The Life Cycle Service defines a *LifeCycleObject* interface. This interface defines remove, copy and move operations.

- The Life Cycle Service has been extended to support compound life cycle operations on graphs of related objects. Compound objects (graphs of objects) rely on the Relationship Service for the definition of object graphs.

1.1.4 Persistent Object Service

- The Persistent Object Service (POS) provides a set of common interfaces to the mechanisms used for retaining and managing the persistent state of objects.
- The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. A major feature of the Persistent Object Service is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where mechanisms useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers do not apply to mainframes.

1.1.5 Transaction Service

- The Transaction Service supports multiple transaction models, including the flat (mandatory in the specification) and nested (optional) models.
- The Object Transaction Service supports interoperability between different programming models. For instance, some users want to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires the object and procedural code to share a single transaction.
- Network interoperability is also supported, since users need communication between different systems, including the ability to have one transaction service interoperate with a cooperating transaction service using different ORBs.
- The Transaction Service supports both implicit (system-managed transaction) propagation and explicit (application-managed) propagation. With implicit propagation, transactional behavior is not specified in the operation’s signature. With explicit propagation, applications define their own mechanisms for sharing a common transaction.
- The Transaction Service can be implemented in a TP monitor environment, so it supports the ability to execute multiple transactions concurrently, and to execute clients, servers, and transaction services in separate processes.

1.1.6 Concurrency Control Service

- The Concurrency Control Service enables multiple clients to coordinate their access to shared resources. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

- Concurrent use of a resource is regulated with locks. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the client's activities might conflict. Hence, a client must obtain an appropriate lock before accessing a shared resource. The Concurrency Control Service defines several lock modes, which correspond to different categories of access. This variety of lock modes provides flexible conflict resolution. For example, providing different modes for reading and writing lets a resource support multiple concurrent clients on a read-only transaction. The Concurrency Control Service also defines Intention Locks that support locking at multiple levels of granularity.

1.1.7 Relationship Service

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: relationships and roles. A role represents a CORBA object in a relationship. The Relationship interface can be extended to add relationship-specific attributes and operations. In addition, relationships of arbitrary degree can be defined. Similarly, the *Role* interface can be extended to add role-specific attributes and operations.
- Type and cardinality constraints can be expressed and checked: exceptions are raised when the constraints are violated.
- The Life Cycle Service defines operations to copy, move, and remove graphs of related objects, while the Relationship Service allows graphs of related objects to be traversed without activating the related objects.
- Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references: role objects can be located with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.
- The Relationship Service supports the compound life cycle component of the Life Cycle Service by defining object graphs.

1.1.8 Externalization Service

- The Externalization Service defines protocols and conventions for externalizing and internalizing objects. Externalizing an object is to record the object state in a stream of data (in memory, on a disk file, across the network, and so forth) and then be internalized into a new object in the same or a different process. The externalized object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and be internalized in a different, disconnected ORB. For portability, clients can request that externalized data be stored in a file whose format is defined with the Externalization Service Specification.
- The Externalization Service is related to the Relationship Service and parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for facilities, directory services, and file services.

1.1.9 Query Service

- The purpose of the Query Service is to allow users and objects to invoke queries on collections of other objects. The queries are declarative statements with predicates and include the ability to specify values of attributes; to invoke arbitrary operations; and to invoke other Object Services.
- The Query Service allows indexing; maps well to the query mechanisms used in database systems and other systems that store and access large collections of objects; and is based on existing standards for query, including SQL-92, OQL-93, and OQL-93 Basic.
- The Query Service provides an architecture for a nested and federated service that can coordinate multiple, nested query evaluators.

1.1.10 Licensing Service

- The Licensing Service provides a mechanism for producers to control the use of their intellectual property. Producers can implement the Licensing Service according to their own needs, and the needs of their customers, because the Licensing Service does not impose it own business policies or practices.
- A license in the Licensing Service has three types of attributes that allow producers to apply controls flexibly: *time*; *value mapping*, and *consumer*. Time allows licenses to have start/duration and expiration dates. Value mapping allows producers to implement a licensing scheme according to units, allocation (through concurrent use licensing), or consumption (for example, metering or allowance of grace periods through "overflow licenses.") Consumer attributes allow a license to be reserved or assigned for specific entities; for example, a license could be assigned to a particular machine. The Licensing Service allows producers to combine and derive from license attributes.
- The Licensing Service consists of a *LicenseServiceManager* interface and a *ProducerSpecificLicenseService* interface: these interfaces do not impose business policies upon implementors.

1.1.11 Property Service

- Provides the ability to dynamically associate named values with objects outside the static IDL-type system.
- Defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples. The names are simple OMG IDL strings. The values are OMG IDL *any*s. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system. The modes are similar to those defined in the *Interface Repository AttributeDef* interface.
- Designed to be a basic building block, yet robust enough to be applicable for a broad set of applications.

- Provides "batch" operations to deal with sets of properties as a whole. The use of "batch" operations is significant in that the systems and network management (SNMP, CMIP, ...) communities have proven such a need when dealing with "attribute" manipulation in a distributed environment.
- Provides exceptions such that *PropertySet* implementors may exercise control of (or apply constraints to) the names and types of properties associated with an object, similar in nature to the control one would have with CORBA attributes.
- Allows *PropertySet* implementors to restrict modification, addition and/or deletion of properties (readonly, fixed) similar in nature to the restrictions one would have with CORBA attributes.
- Provides client access and control of constraints and property modes.
- Does not rely on any other object services.

1.1.12 Time Service

- Enables the user to obtain current time together with an error estimate associated with it.
- Ascertains the order in which "events" occurred.
- Generates time-based events based on timers and alarms.
- Computes the interval between two events.
- Consists of two services, hence defines two service interfaces:
  - Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.
  - Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

1.1.13 Security Service

The security functionality defined by this specification comprises:

- **Identification** and **authentication** of principals (human users and objects which need to operate under their own rights) to verify they are who they claim to be.
- **Authorization** and **access control** - deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes) can access it.
- **Security auditing** to make users accountable for their security related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.

- **Security of communication** between objects, which is often over insecure lower layer communications. This requires trust to be established between the client and target, which may require **authentication of clients to targets** and **authentication of targets to clients**. It also requires **integrity protection** and (optionally) **confidentiality protection** of messages in transit between objects.
- **Non-repudiation** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.
- **Administration** of security information (for example, security policy) is also needed.

1.1.14 Object Trader Service

The Object Trader Service provides a matchmaking service for objects.

The Service Provider registers the availability of the service by invoking an export operation on the trader, passing as parameters information about the offered service. The export operation carries an object reference that can be used by a client to invoke operations on the advertised services, a description of the type of the offered service (i.e., the names of the operations to which it will respond, along with their parameter and result types), information on the distinguishing attributes of the offered service.

The offer space managed by traders may be partitioned to ease administration and navigation. This information is stored persistently by the Trader. Whenever a potential client wishes to obtain a reference to a service that does a particular job, it invokes an import operation, passing as parameters a description of the service required. Given this import request, the Trader checks appropriate offers for acceptability. To be acceptable, an offer must have a type that conforms to that requested and have properties consistent with the constraints specified by an imported.

Trading service in a single trading domain may be distributed over a number of trader objects. Traders in different domains may be federated. Federation enables systems in different domains to negotiate the sharing of services without losing control of their own policies and services. A domain can thus share information with other domains with which it has been federated, and it can now be searched for appropriate service offers.

1.1.15 Object Collections Service

Collections are groups of objects which, as a group, support some operations and exhibit specific behaviors that are related to the nature of the collection rather than to the type of object they contain. Examples of collections are sets, queues, stacks, lists, binary, and trees. The purpose of the Collection Object Service is to provide a uniform way to create and manipulate the most common collections generically.

Examples of collections are sets, queues, stacks, lists, binary, and trees. For example, sets might support the following operations: insert new element, membership test, union, intersection, cardinality, equality test, emptiness test, etc. One of the defining semantics of a set is that, if an object O is a member of a set S, then inserting O into S results in the set being unchanged. This property would not hold for another collection type called a bag.

General Design Principles 2

This chapter discusses the principles that were considered in designing Object Services and their interfaces. It also addresses dependencies between Object Services, their relationship to CORBA, and their conformance to existing standards.

2.1 Service Design Principles

2.1.1 Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

2.1.2 Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as "building blocks."

2.1.3 Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

2.1.4 Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

2.1.5 Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

2.1.6 Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer*, *PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. This is shown graphically in Figure 2-1.

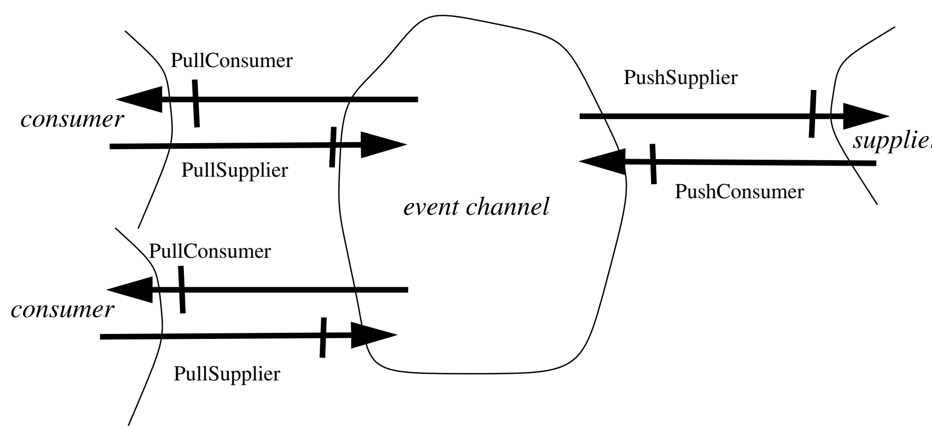


Figure 2-1 An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

The graphical notation shown in Figure 2-1 is used throughout this document and in the full service specifications. An arrow with a vertical bar is used to show that the target object supports the interface named below the arrow and that clients holding an object reference to it of this type can invoke operations. In shorthand, one says that the object reference (held by the client) supports the interface. The arrow points from the client to the target (server) object.

A blob (misshapen circle) delineates a conspiracy of one or more objects. In other words, it corresponds to a conceptual object that may be composed of one or more CORBA objects that together provide some coordinated service to potentially multiple clients making requests using different object references.

2.1.7 Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms

2.1.8 Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

2.1.9 Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

2.2 Interface Style Consistency

2.2.1 Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

2.2.2 Explicit Versus Implicit Operations

Operations are always explicit rather than implied e.g. by a flag passed as a parameter value to some “umbrella” operation. In other words, there is always a distinct operation corresponding to each distinct function of a service.

2.2.3 Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by “normal” clients.

2.3 Key Design Decisions

2.3.1 Naming Service: Distinct from Property and Trading Services

The Naming Service is addressed separately from property and trading services.

Naming contexts have some similarity to property lists (that is, lists of values associated with objects though not necessarily part of the object’s state). The Naming Service in general also has elements in common with a trading service. However, following the “Bauhaus” principle of keeping services as simple and as orthogonal as possible, these services have been kept distinct and are being addressed separately.

2.3.2 Universal Object Identity

The services described in this manual do not require the concept of object identity.

2.4 Integration with Future Object Services

This section discusses how the Object Services could evolve to integrate with future services, such as:

- Archive
- Backup/Restore
- Change Management (Versioning)
- Data Interchange
- Implementation Repository
- Internationalization
- Logging
- Recovery
- Replication
- Startup

2.4.1 Archive Service

Persistent Object Service. The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service should be able to archive objects stored with the Persistent Object Service.

Externalization Service. The Archive Service copies objects from an active/persistent store to a backup store and vice versa. This service could use the Externalization Service to get the internal state of objects for saving and to subsequently recreate objects with this stored state. If only persistent objects need to be archived, then the Object Persistence Service could be used instead.

2.4.2 Backup/Restore Service

Externalization Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Object Externalization Service as an underlying mechanism for objects regardless of whether they are persistent.

Persistent Object Service. The Backup/Restore Service provides recovery after a system failure or a user error. This service could use the Persistent Object Service as an underlying mechanism for persistent objects.

Transaction Service. The permanence of effect property of a transaction implies that the state established by the commitment of a transaction will not be lost. To guarantee this property, the storage media on which the objects updated by the transaction are stored must be backed-up to secondary storage to ensure that they are not lost should the primary storage media fail. Similarly, the storage media used by the logging service must be restorable should the media fail. Since there are multiple components which require backup services, a single interface would be advantageous.

2.4.3 Change Management Service

Persistent Object Service. The Change Management Service supports the identification and consistent evolution of objects including version and configuration management. This service should work with the Persistent Object Service to allow persistent objects to evolve from the old to new versions.

2.4.4 Data Interchange Service

Persistent Object Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service should work with Persistent Object Service to allow state to be exchanged when one or more of the objects are persistent.

2.4.5 Internationalization Service

Naming Service. Naming Service interfaces may also need to be extended (for example, the structure of names extended, additional name resolution operations added) to better support representing and resolving names for some languages and cultures.

2.4.6 Implementation Repository

Persistent Object Service. The Implementation Repository supports the management of object implementations. The Persistent Object Service may depend on this to determine what persistent data an object contains. This dependency is at the implementation level.

2.4.7 Interface Repository

Persistent Object Service. The Interface Repository supports runtime access to OMG IDL-specified definitions such as object interfaces and type definitions. The Persistent Object Service depends on this to determine if a persistent object supports certain interfaces.

2.4.8 Logging Service

Transaction Service. A logging service implements the abstract notion of an infinitely long, sequentially-accessible, append-only file. It typically supports multiple log files, where each log file consists of a sequence of log records. New log records are written to the end of a log file, old log records can be read from any position in the file. To stop log files from growing too large for the underlying storage medium, a log service must provide an operation to archive old log records to allow the log file to be truncated.

Various components of a transaction processing system may require the services of a log service:

- Transaction Service: during the two-phase commit protocol the Transaction Service must log its state to ensure that the outcome of the committing transaction can be determined should there be a failure.
- Recoverable (transactional) objects: a log can be used to record old and new versions of a recoverable object for the purposes of supporting recovery.
- Locking service: a log can be used to record the locks held on an object at prepare time to facilitate recovery.

Since there are multiple components within a distributed transaction processing system that require the services of a log service, a single log service interface (and potentially server) that is shared between the components is clearly advantageous.

The correctness of a transaction service depends upon the services of a log service, for this reason, the log service must meet the following requirements:

1. Restart.

   A restart facility allows rapid recovery from the cold start of an application. The recovery service used by the application (indirectly through the application's use of recoverable objects) would use the restart facility to establish a *checkpoint*: a consistent point in the execution state of the application from which the recovery process can proceed. In the absence of a checkpoint the recovery service would have to scan the entire log to ensure restart recovery occurs correctly.

2. Buffering and forcing operations.

   A log service should provide two classes of operation for writing log records:

   a. An operation to buffer a log record (the record is not written directly to the underlying storage medium). Used during the execution of a transaction. Since the log record is buffered the write is inexpensive.

   b. An operation to force a log record to the underlying storage medium. Used during the two-phase commit protocol to guarantee the correctness of the transaction. Forcing a log record also flushes all previously written, but buffered, log records.

3. Robustness.

   The log service should ensure the consistency of the underlying storage medium in which log files are stored. This usually involves the log service employing protocols that update the storage in a manner that would not result in the loss of any existing data (i.e. careful updates), along with support for mirroring the storage media to tolerate media failures.

4. Archival.

   A log service should provide support for archiving log records. Archival is necessary to allow the log to be truncated to ensure that it does not grow without bounds.

5. Efficiency.

   Since the log service may be written to by multiple components within a transaction, the addition of log records must be efficient to avoid the bandwidth of log from becoming a bottleneck in the system.

2.4.9 Recovery Service

Transaction Service. As recoverable objects are updated during a transaction, they (as resource managers) keep a record of the changes made to their state that is sufficient to undo the updates should the transaction rollback. The component responsible for this task is termed the recovery service. Various different forms of recovery are possible, however the most common form is called value logging and involves the recoverable object recording both the old and new values of the object. When a transaction is recovered due to failure, the old value of an object is used to undo changes made to the object during the transaction. Most recovery services employ the services of a logging service (described in this section) to maintain the "undo" information. The definition of a standard recovery service interface is one possible additional CORBA-compliant object service.

2.4.10 Replication Service

Persistent Object Service. The Replication Service provides explicit replication of objects in a distributed environment and manages the consistency of replicated copies. This service could use the Persistent Object Service to manage persistent replicas.

2.4.11 Startup Service

Persistent Object Service. The Startup Service supports bootstrapping and termination of the Persistent Object Service.

2.4.12 Data Interchange Service

Externalization Service. The Data Interchange Service enables objects to exchange some or all of their associated state. This service could use the Object Externalization Service to allow state to be exchanged regardless of whether the objects are persistent.

2.5 Service Dependencies

The interface designs of all the services are general in nature and do not presume or require the existence of specific supporting software in order to implement them. An implementation of the Name Service, for instance, could use naming or directory services provided in a general-purpose networking environment. For example, an implementation may be based on the naming services provided by ONC or DCE. Such an implementation could provide enterprise-wide naming services to both object-based and non-object-based clients. Object-based software would see such services through the use of NamingContext objects.

Although the Object Services do not depend upon specific software, some dependencies and relationships do exist between services.

2.5.1 Event Service

The Event Service does not depend upon other services.

2.5.2 Life Cycle Service

Interfaces for the Life Cycle Service depend on the Naming Service.

The Life Cycle Service also defines compound operations that depend on the Relationship Service for the definition of object graphs. Appendix A describes the topic of compound life cycle, and its dependence on the Relationship Service, in detail.

2.5.3 Persistent Object Service

The Externalization Service provides functions that provide for the transformation of an object into a form suitable for storage on an external media or for transfer between systems. The Persistent Object Service uses this service as a POS protocol.

The Life Cycle Service provides operations for managing object creation, deletion, copy and equivalence. The Persistent Object Service depends on this service for creating and deleting all required objects.

The Naming Service provides mappings between user-comprehensible names and CORBA object references. The Persistent Object Service depends on this service to obtain the object reference of, say, a PDS from its name or id.

2.5.4 Relationship Service

The Relationship Service does not depend on other services. Note especially that the Relationship Service does not depend on any common storage service.

For guidelines about when to use the Relationship Service and when to use CORBA object references, refer to the section "The Relationship Service vs CORBA Object References," in Chapter 9.

2.5.5 Externalization Service

The Externalization Service works with the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. Specifically, this service uses the Life Cycle Service to create and remove Stream and StreamFactory objects. ORB services may be used in Stream implementations to identify InterfaceDef and ImplementationDef objects corresponding to an externalized object, and to support finding an appropriate factory for recreating that object at internalization time.

The Externalization Service can also work with the Relationship Service. Implementations of Stream and StreamIO operations could use the Relationship Service to ensure that multiple references to the same object or circular references don't result in duplication of objects at internalization time or in the external representation.

In addition, the Externalization Service adds compound externalization semantics to the containment and reference relationships in the Relationship Service. Detailed information is provided in "Specific Externalization Relationships" on page 8-26.

2.5.6 Transaction Service

As concurrent requests are processed by an object a mechanism is required to mediate access. This is necessary to provide the transaction property of isolation. The Concurrency Control Service is one possible implementation of a locking service.

The Transaction Service depends upon the Concurrency Control Service in the following ways:

- Concurrency Control Service must support transaction duration locks, which provide isolation of concurrent requests by different transactions.
- Concurrency Control Service must record transaction duration locks on persistent media, such as a log, as part of the prepare phase of commitment.
- If nested transactions are supported by the Transaction Service then the Concurrency Control Service must also support locks that provide isolation between siblings in a transaction family and provide inheritance of locks owned by a subransaction to its parent when the subtransaction commits.

- Transactional clients of the Concurrency Control Service are responsible for ensuring that all locks held by a transaction are dropped after all recovery or commitment operations have taken place. The drop-licks operation is provided by the LockCoordinator interface for this purpose.

The Transaction Service supports atomicity and durability properties through the Persistent Object Service (POS). The Transaction Service can work with the POS to support atomic execution of operations on persistent objects. Transactions and persistence are not provided by the same service. When coordination of multiple state changes are required to persistent data, a persistence service requires a transaction service. The POS can provide persistence, but its implementation needs to be changed to support transactional behavior. There are no changes to the interfaces of the POS to support transactions. The following discussion applies to support of persistence when a transaction service is required.

Support for persistence can be built from other specialized services that can also be shared by other object services. Examples include:

- Recovery service: this supports the atomicity and durability properties.
- Logging service: this is used by the recovery service to assist in supporting the atomicity and durability properties. It is also used by the Transaction Service to support the two-phase commit protocol.
- Backup and restore service: this supports the isolation property.

This view is consistent with the X/Open DTP (Distributed Transaction Processing) model which separates the transaction manager service (i.e. the implementation of a generalized two-phase commit protocol) from a resource manager that provides services for data with a life beyond process execution. This permits both transactions on transient objects and on persistent objects without transactions.

2.5.7 Concurrency Control Service

The Concurrency Control Service does not depend on any other service per se. Nevertheless, it is designed to work with the Transaction Service.

2.5.8 Query Service

The Query Service does not depend on other service but is closely related to these Object Services: Life Cycle; Persistent Object; Relationship; Concurrency Control; Transaction; Property; and Collection.

2.5.9 Licensing Service

The Licensing Service depends on the Event Service. It may depend on the Relationship, Property, and Query Services for some implementations. This dependency is determined by an implementation's policy definition and entry capability. The Licensing Service also depends on the Security Service, because the Licensing Service interface can use unforgeable and secure events. The Licensing Service will use Security Service interfaces to support the requirements addressed by the challenge mechanism.

2.5.10 Property Service

The Property Service does not depend upon other services; however, it is closely related to Collection Service.

2.5.11 Time Service

The Time Service does not depend upon other services.

2.5.12 Security Service

The Security Service does not depend upon other services.

2.5.13 Trader Service

The Trader Service does not depend upon other services.

2.5.14 Collections Service

The Collections Service does not depend upon other services; however, it is closely related to these services: Concurrency, Naming, Persistent Object, Property, and Query.

2.6 Relationship to CORBA

This section provides information about the relationship of other services to the CORBA specification.

2.6.1 ORB Interoperability Considerations: Transaction Service

Some implementations of the Transaction Service will support:

- The ability of a single application to use both object and procedural interfaces to the Transaction Service. This is described as part of the specification, particularly in the sections "The User's View" and 'The Implementor's View."
- The ability for different Transaction Service implementations to interoperate across a single ORB. This is provided as a consequence of this specification, which defines IDL interfaces for all interactions between Transaction Service implementations.
- The ability for the same Transaction Service to interoperate with another instance of itself across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)
- The ability for different Transaction Services implementations to interoperate across different ORBs. (This ability is supported by the Interoperability specification of CORBA 2.0.)

- A critical dependency for Transaction Service interoperation across different ORBs is the handling of the propagation_context between ORBs. This includes the following:
  - Efficient transformation between different ORB representations of the propagation_context.
  - The ability to carry the ID information (typically an X/Open XID) between interoperating ORBs.
  - The ability to do interposition to ensure efficient local execution of the is_same_transaction operation.

2.6.2 Life Cycle Service

The Life Cycle Service assumes CORBA implementations support object relocation.

2.6.3 Naming Service

Entities that are not CORBA objects - that is to say, not objects accessed via an Object Request Broker - are used for names (in the guise of pseudo objects). In both cases the interfaces to these entities conform as closely as possible to OMG IDL while satisfying the specific service design requirements, in order to enable maximum flexibility in the future. Specifically, in the Naming Service, name objects are pseudo objects with interfaces defined in pseudo IDL (PIDL). These objects look like CORBA objects but are specifically designed to be accessed using a programming language binding. This is done for reasons based on the expected use of these objects.

2.6.4 Relationship Service

The Relationship Service requires CORBA Interface Repositories to support the ability to dynamically determine if an InterfaceDef conforms to another InterfaceDef, that is, if it is a subtype. This is needed to implement type constraints for particular relationships.

2.6.5 Persistent Object Service

The Persistent Object Service requires CORBA Interface Repositories.

2.6.6 General Interoperability Requirements

Interoperability between Object Services and users of Object Services implemented on different ORBs requires common RepositoryIDs be used to identify types in both systems. The types identified by these RepositoryIDs must also be consistently defined. As described in Common Object Request Broker: Architecture and Specification, Pragma Directives for Repository Id section, all CORBAservice IDL presented in this specification is implicitly preceded at file scope by the following directive:

```
#pragma prefix “omg.org”
```

Object Service Implementations that choose to extend the standard interfaces must do so by deriving new interfaces rather than by modifying the standard interfaces.

2.7 Relationship to Object Model

All specifications contained in this manual conform to the OMG Object Model. No additional components or profiles are required by any service.

2.8 Conformance to Existing Standards

In general, existing relevant standards do not have object-oriented interfaces nor are they structured in a form that is easily mapped to objects. These specifications have been influenced by existing standards, and services have been designed which minimize the difficulty of encapsulating supporting software. The naming service specification is believed to be compatible with X.500, DCE CDS and ONC NIS and NIS+.

These specifications are broadly conformant to emerging ISO/IEC/CCITT ODP standards:

- CCITT Draft Recommendations X.900, ISO/IEC 10746 Basic Reference Model for Open Distributed Computing
- ISO/IEC JTC1 SC21 WG7 N743 Working Document on Topic 9.1 - ODP Trader

Naming Service Specification 3

3.1 Service Description

3.1.1 Overview

A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To *resolve a name* is to determine the object associated with the name in a given context. To *bind a name* is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names.

Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a *naming graph* — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a *compound name*) defines a path in the naming graph to navigate the resolution process. Figure 3-1 shows an example of a naming graph.

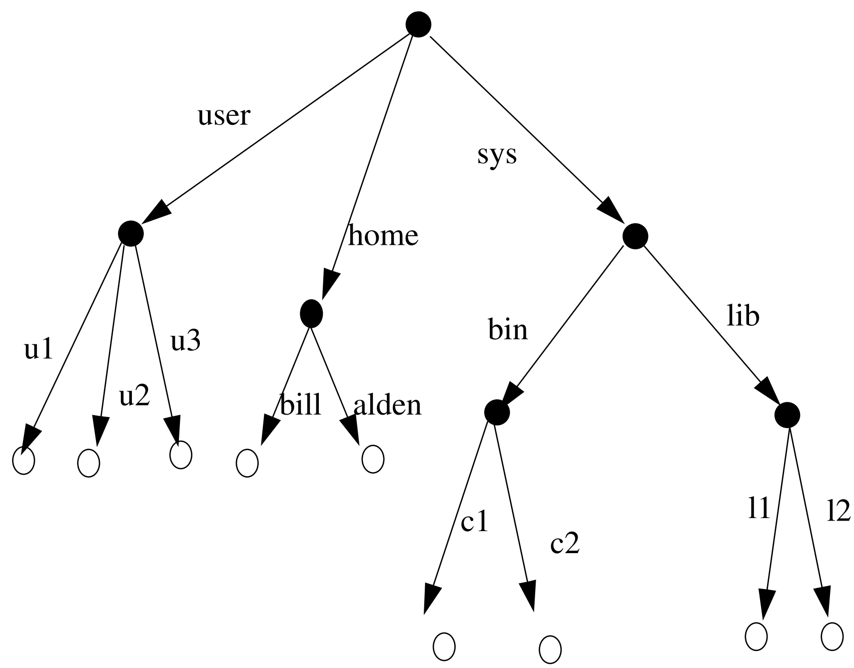


Figure 3-1 A Naming Graph

3.1.2 Names

Many of the operations defined on a naming context take names as parameters. Names have structure. A name is an ordered sequence of *components*.

A name with a single component is called a *simple name*; a name with multiple components is called a *compound name*. Each component except the last is used to name a context; the last component denotes the bound object. The notation:

< component1 ; component2 ; component3 >

indicates the sequences of components.

Note – The semicolon (;) characters are simply the notation used in this document and are not intended to imply that names are sequences of characters separated by semicolon.

A name component consists of two attributes: the *identifier attribute* and the *kind attribute*. Both the identifier attribute and the kind attribute are represented as IDL strings.

The kind attribute adds descriptive power to names in a syntax-independent way. Examples of the value of the kind attribute include *c_source*, *object_code*, *executable*, *postscript*, or " ". The naming system does not interpret, assign, or manage these values in any way. Higher levels of software may make policies about the use and management of these values. This feature addresses the needs of applications that use syntactic naming conventions to distinguish related objects. For example Unix uses suffixes such as .c and .o. Applications (such as the C compiler) depend on these syntactic convention to make name transformations (for example, to transform foo.c to foo.o).

The lack of name syntax is especially important when considering internationalization issues. Software that does not depend on the syntactic conventions for names does not have to be changed when it is localized for a natural language that has different syntactic conventions — unlike software that does depend on the syntactic conventions (which must be changed to adopt to new conventions).

To avoid issues of differing name syntax, the Naming Service always deals with names in their structural form (that is, there are no canonical syntaxes or distinguished meta characters). It is assumed that various programs and system services will map names from the representation into the structural form in a manner that is convenient to them.

3.1.3 Names Library

To allow the representation of names to evolve without affecting existing clients, it is desirable to hide the representation from client code. Ideally, names themselves would be OMG IDL objects; however, names must be lightweight entities that can be very efficiently created and manipulated in memory and passed as parameters in requests by value. In order to simplify name manipulation and provide representation independence, names can be presented to programs through the names library. Note, however, it is not necessary to use the names library to use the basic operations of the naming service.

The names library implements names as pseudo-objects. A client makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL. The names library supports two pseudo-IDL interfaces: the *LNameComponent* interface and the *LName* interface. The *LNameComponent* interface defines the get and set operations associated with name component identifier and the kind attributes.The *LName* Interface includes operations for manipulating library name and library name component pseudo objects and producing and translating a structure that can be passed as a parameter to a normal object request.

3.1.4 Example Scenarios

This section provides two short scenarios that illustrate how the naming service specification can be used by two fairly different kinds of systems -- systems that differ in the kind of implementations used to build the Naming Service and that differ in models of how clients might use the Naming Service with other object services to locate objects.

In one system, the Naming Service is implemented using an underlying enterprise-wide naming server such as DCE CDS. The Naming Service is used to construct large, enterprise-wide naming graphs where NamingContexts model "directories" or "folders" and other names identify "document" or "file" kinds of objects. In other words, the naming service is used as the backbone of an enterprise-wide filing system. In such a system, non-object-based access to the naming service may well be as commonplace as object-based access to the naming service. For example, the name of an object might be presented to the DCE directory service as a null-terminated ASCII string such as “/.../DME/nls/moa-1/ID-1”.

The Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of). Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain "externally visible" characteristics, for example, for objects with recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

Conventions are employed in such a scheme that meaningfully partition the name space. For example, individuals are assigned naming contexts for personal use, groups of individuals may be assigned shared naming contexts while other contexts are organized in a public section of the naming graph. Similarly, conventions are used to identify contexts that list the names of services that are available in the system (e.g., that locate a translation or printing service).

In an alternative system, the Naming Service can be used in a more limited role and can have a less sophisticated implementation. In this model, naming contexts represent the types and locations of services that are available in the system and a much shallower naming graph is employed. For example, the Naming Service is used to register the object references of a mail service, an information service, a filing service.

Given a handful of references to "root objects" obtained from the Naming Service, a client uses the Relationship and Query Services to locate objects contained in or managed by the services registered with the Naming Service. In such a system, the Naming Service is used sparingly and instead clients rely on other services such as query services to navigate through large collections of objects. Also, objects in this scheme rarely register "external characteristics" with another service - instead they support the interfaces of Query or Relationship Services.

Of course, nothing precludes the Naming Service presented here from being used to provide both models of use at the same time. These two scenarios demonstrate how this specification is suitable for use in two fairly different kinds of systems with potentially quite different kinds of implementations. The service provides a basic building block on which higher-level services impose the conventions and semantics which determine how frameworks of application and facilities objects locate other objects.

3.1.5 Design Principles

Several principles have driven the design of the Naming Service:

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software. The naming service provides only a structural convention for names, e.g. compound names.

2. The design supports distributed, heterogeneous implementation and administration of names and name contexts.

3. Names are structures, not just character strings. A struct is necessary to avoid encoding information syntactically in the name string (e.g., separating the human-meaningful name and its type with a ".", and the type and version with a "!"), which is a bad idea with respect to the generality, extensibility, and internationalization of the name service. The structure define includes a human-chosen string plus a kind field.

4. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.

5. The Naming Service is a fundamental object service, with no dependencies on other interfaces.

6. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No "universal" root is needed for a name hierarchy.

7. Existing name and directory services employed in different network computing environments can be transparently encapsulated using name contexts. All of the above features contribute to making this possible.

8. The design does not address name security since there is currently no OMG security model. The Naming Service can be evolved to provide name security when an object security service is standardized.

9. The design does not address namespace administration. It is the responsibility of higher-level software to administer the namespace.

3.1.6 Resolution of Technical Issues

This specification addresses the issues identified for a name service in the OMG *Object Services Architecture* document[1] as follows:

- *Naming standards*: Encapsulation of existing naming standards and protocols is allowed using naming contexts. Transparent encapsulation is made possible by the design features outlined above.

1. *Object Services Architecture*, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

- *Federation of namespaces*: The specification supports distributed federation of namespaces; no assumptions are made about centralized or universal functions. Namespaces may be nested in a graph in any fashion, independent of the implementation of each namespace. There need be no distinguished root context, and existing graphs may be joined at any point.
- *Scope of names*: Name contexts define name scope. Names must be unique within a context. Objects may have multiple names, and may exist in multiple name contexts. Name contexts may be named objects nested within another name context, and cycles are permitted. The name itself is not a full-fledged ORB object, but does support structure, so it may have multiple components. No requirements are placed on naming conventions, in order to support a wide variety of conventions and existing standards.
- *Operations*: The specification supports bind, unbind, lookup, and sequence operations on a name context. It does not support a rename operation, because we do not see how to implement this correctly in a distributed environment without transactions.

3.2 The CosNaming Module

The CosNaming Module is a collection of interfaces that together define the naming service. This module contains two interfaces:

- The *NamingContext* interface
- The *BindingIterator* interface

This section describes these interfaces and their operations in detail.

The CosNaming Module is shown in Figure 3-2. Note that Istring is a placeholder for a future IDL internationalized string data type.

```
module CosNaming
{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };

    typedef sequence <NameComponent> Name;

    enum BindingType {nobject, ncontext};

    struct Binding {
        Name binding_name;
        BindingType binding_type;
```

Figure 3-2 The CosNaming Module

```
};

typedef sequence <Binding> BindingList;


interface BindingIterator;

interface NamingContext {

    enum NotFoundReason { missing_node, not_context, not_object};

    exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
    };

    exception CannotProceed {
            NamingContext cxt;
            Name rest_of_name;
    };

    exception InvalidName{};
    exception AlreadyBound {};
    exception NotEmpty{};

    void bind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName);
    void bind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName);
    Object resolve (in Name n)
        raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n)
        raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context();
    NamingContext bind_new_context(in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
    void destroy( )
        raises(NotEmpty);
    void list (in unsigned long how_many,
                out BindingList bl, out BindingIterator bi);
};

interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
```

Figure 3-2 The CosNaming Module *(Continued)*

```
                        out BindingList bl);
            void destroy();
        };
};
```

Figure 3-2 The CosNaming Module *(Continued)*

The following sections describe the operations of the *NamingContext* interface:

- binding objects
- name resolution
- unbinding
- creating naming contexts
- deleting contexts
- listing a naming context

3.2.1 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the resolve operation. The Naming Service supports four operations to create bindings: *bind*, *rebind*, *bind_context* and *rebind_context*.

```
void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
    raises(NotFound, CannotProceed, InvalidName);
```

bind

Creates a binding of a name and an object in the naming context. Naming contexts that are bound using bind do not participate in name resolution when compound names are passed to be resolved.

A bind operation that is passed a compound name is defined as follows:

ctx->bind(< c1 ; c2 ; ... ; cn >, obj) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind(< cn >, obj)

rebind

Creates a binding of a name and an object in the naming context even if the name is already bound in the context. Naming contexts that are bound using rebind do not participate in name resolution when compound names are passed to be resolved.

bind_context
Names an object that is a naming context. Naming contexts that are bound using bind_context() participate in name resolution when compound names are passed to be resolved.

A bind_context operation that is passed a compound name is defined as follows:

ctx->bind_context(< c1 ; c2 ; ... ; cn >, nc) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind_context(< cn >, nc)

rebind_context
Creates a binding of a name and a naming context in the naming context even if the name is already bound in the context. Naming contexts that are bound using rebind_context() participate in name resolution when compound names are passed to be resolved.

Table 3-1 describes the exceptions raised by the binding operations.

Table 3-1 Exceptions Raised by Binding Operations

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)
AlreadyBound	Indicates an object is already bound to the specified name. Only one object can be bound to a particular name in a context. The bind and the bind_context operations raise the AlreadyBound exception if the name is bound in the context; the rebind and rebind_context operations unbind the name and rebind the name to the object passed as an argument.

3.2.2 Resolving Names

The resolve operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for “narrowing” the object to the appropriate type. That is, clients typically cast the returned object from Object to a more specialized interface. The OMG IDL definition of the resolve operation is:

```
Object resolve (in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. A compound resolve is defined as follows:

ctx->resolve(< c1 ; c2 ; ... ; cn >) ≡
ctx->resolve(< c1 ; c2 ; ... ; cn-1 >)->resolve(< cn >)

Table 3-2 describes the exceptions raised by the resolve operation.

Table 3-2 Exceptions Raised by Resolve Operation

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)

3.2.3 Unbinding Names

The unbind operation removes a name binding from a context. The definition of the unbind operation is:

```
void unbind(in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

A unbind operation that is passed a compound name is defined as follows:

ctx->unbind(< c1 ; c2 ; ... ; cn >) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->unbind(< cn >)

Table 3-3 describes the exceptions raised by the unbind operation.

Table 3-3 Exceptions Raised by Unbind Operation

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)

3.2.4 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: *new_context* and *bind_new_context*.

```
NamingContext new_context();

NamingContext bind_new_context(in Name n)
raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

new_context
: This operation returns a naming context implemented by the same naming server as the context on which the operation was invoked. The new context is not bound to any name.

bind_new_context
: This operation creates a new context and binds it to the name supplied as an argument. The newly-created context is implemented by the same naming server as the context in which it was bound (that is, the naming server that implements the context denoted by the name argument excluding the last component).

A bind_new_context that is passed a compound name is defined as follows:

ctx->bind_new_context(< c1 ; c2 ; ... ; cn >) ≡
(ctx->resolve(< c1 ; c2 ; ... ; cn-1 >))->bind_new_context(< cn >)

Table 3-4 describes the exceptions raised when new contexts are being created.

Table 3-4 Exceptions Raised by Creating New Contexts

Exception Raised	Description
NotFound	Indicates the name does not identify a binding.
CannotProceed	Indicates that the implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.
InvalidName	Indicates the name is invalid. (A name of length 0 is invalid; implementations may place other restrictions on names.)
AlreadyBound	Indicates an object is already bound to the specified name. Only one object can be bound to a particular name in a context.

3.2.5 Deleting Contexts

The destroy operation deletes a naming context:.

```
void destroy()
    raises(NotEmpty);
```

If the naming context contains bindings, the NotEmpty exception is raised.

3.2.6 Listing a Naming Context

The list operation allows a client to iterate through a set of bindings in a naming context.

```
enum BindingType {object, ncontext};

struct Binding {
        Name        binding_name;
        BindingType binding_type;
};

typedef sequence <Binding> BindingList;

void list (in unsigned long how_many,
                        out BindingList bl, out BindingIterator bi);
};
```

The list operation returns at most the requested number of bindings in BindingList bl.

- If the naming context contains additional bindings, the list operation returns a BindingIterator with the additional bindings.
- If the naming context does not contain additional bindings, the binding iterator is a nil object reference.

3.2.7 The BindingIterator Interface

The *BindingIterator* interface allows a client to iterate through the bindings using the next_one or next_n operations:

```
interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many,
                        out BindingList bl);
        void destroy();
};
```

next_one
: This operation returns the next binding. If there are no more bindings, *false* is returned.

next_n
: This operation returns at most the requested number of bindings.

destroy
: This operation destroys the iterator.

3.3 The Names Library

To allow the representation of names to evolve without affecting existing clients, it is desirable to hide the representation of names from client code. Ideally, names themselves would be objects; however, names must be lightweight entities that are efficient to create, manipulate, and transmit. As such, names are presented to programs through the *names library.*

The names library implements names as *pseudo-objects*. A client makes calls on a pseudo-object in the same way it makes calls on an ordinary object. Library names are described in pseudo-IDL (to suggest the appropriate language binding). C and C++ clients[2] use the same client language bindings for pseudo-IDL (PIDL) as they use for IDL.

Pseudo-object references cannot be passed across OMG IDL interfaces. As described in Section 3.2, "The CosNaming Module," the naming service supports the *NamingContext* OMG IDL interface. The names library supports an operation to convert a library name into a value that can be passed to the name service through the *NamingContext* interface.

Note – It is not a requirement to use the names library in order to use the Naming Service.

The names library consists of two pseudo-IDL interfaces: the *LNameComponent* interface and the *LName* interface, as shown in Figure 3-3.

2. As anticipated

```
interface LNameComponent {                    // PIDL
        exception NotSet{};
        string get_id()
                raises(NotSet);
        void set_id(in string i);
        string get_kind()
                raises(NotSet);
        void set_kind(in string k);
        void destroy();
};

interface LName {                                              // PIDL
        exception NoComponent{};
        exception OverFlow{};
        exception InvalidName{};
        LName insert_component(in unsigned long i,
                                in LNameComponent n)
                        raises(NoComponent, OverFlow);
        LNameComponent get_component(in unsigned long i)
                 raises(NoComponent);
        LNameComponent delete_component(in unsigned long i)
                 raises(NoComponent);
        unsigned long num_components();
        boolean equal(in LName ln);
        boolean less_than(in LName ln);
        Name to_idl_form()
                        raises(InvalidName);
        void from_idl_form(in Name n);
        void destroy();
};

LName create_lname();                                          // C   /C++
LNameComponent create_lname_component(); // C/C++
```

Figure 3-3 The Names Library Interface in PIDL

3.3.1 Creating a Library Name Component

To create a library name component pseudo-object, use the following C/C++ function:

```
LNameComponent create_lname_component();                       // C/C++
```

The returned pseudo-object can then be operated on using the operations in Figure 3-3.

3.3.2 Creating a Library Name

To create a library name pseudo-object, use the following C/C++ function.

```
LName create_lname();                                   // C/C++
```

The returned pseudo-object reference can then be operated on using the operations in Figure 3-3.

3.3.3 The LNameComponent Interface

A name component consists of two attributes: the identifier attribute and the kind attribute. The *LNameComponent* interface defines the operations associated with these attributes.

```
string get_id()
        raises(NotSet);
void set_id(in string k);
string get_kind()
        raises(NotSet);
void set_kind(in string k);
```

get_id
: The get_id operation returns the identifier attribute's value. If the attribute has not been set, the NotSet exception is raised.

set_id
: The set_id operation sets the identifier attribute to the string argument.

get_kind
: The get_kind operation returns the kind attribute's value. If the attribute has not been set, the NotSet exception is raised.

set_kind
: The set_kind operation sets the kind attribute to the string argument.

3.3.4 The LName Interface

The following operations are described in this section:

- destroying a library name component pseudo object
- creating a library name
- inserting a name component
- getting the i^{th} name component
- deleting a name component
- number of name components

- testing for equality
- testing for order
- producing an idl form
- translating an idl form
- destroying a library name pseudo object

Destroying a Library Name Component Pseudo Object

The destroy operation destroys library name component pseudo-objects.

```
void destroy();
```

Inserting a Name Component

A name has one or more components. Each component except the last is used to identify names of subcontexts. (The last component denotes the bound object.) The insert_component operation inserts a component after position *i*.

```
LName insert_component(in unsigned long i, in LNameComponent lnc)
        raises(NoComponent, OverFlow);
```

If component *i*-1 is undefined and component *i* is greater than 1, the insert_component operation raises the NoComponent exception.

If the library cannot allocate resources for the inserted component, the Overflow exception is raised.

Getting the i^{th} Name Component

The get_component operation returns the i^{th} component. The first component is numbered *1*.

```
LNameComponent get_component(in unsigned long i)
        raises(NoComponent);
```

If the component does not exist, the NoComponent exception is raised.

Deleting a Name Component

The delete_component operation removes and returns the ith component.

```
LNameComponent delete_component(in unsigned long i)
        raises(NoComponent);
```

If the component does not exist, the NoComponent exception is raised.

After a delete_component operation has been performed, the compound name has one fewer component and components previously identified as *i+1...n* are now identified as *i...n-1*.

Number of Name Components

The num_components operation returns the number of components in a library name.

```
unsigned long num_components();
```

Testing for Equality

The equal operation tests for equality with library name ln.

```
boolean equal(in LName ln);
```

Testing for Order

The less_than operation tests for the order of a library name in relation to library name ln.

```
boolean less_than(in LName ln);
```

This operation returns *true* if the library name is less than the library name ln passed as an argument. The library implementation defines the ordering on names.

Producing an IDL form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo object; therefore, it cannot be passed across the IDL interface for the naming service. Several operations in the *NamingContext* interface have arguments of an IDL-defined structure, Name. The following PIDL operation on library names produces a structure that can be passed across the IDL request.

```
Name to_idl_form()
        raises(InvalidName);
```

If the name is of length 0, the InvalidName exception is returned.

Translating an IDL Form

Pseudo-objects cannot be passed across OMG IDL interfaces. The library name is a pseudo object; therefore, it cannot be passed across the IDL interface for the naming service. The *NamingContext* interface defines operations that return an IDL struct of type Name. The following PIDL operation on library names sets the components and kind attribute for a library name from a returned IDL defined structure, Name.

```
void from_idl_form(in Name n);
```

Destroying a Library Name Pseudo-Object

The destroy operation destroys library name pseudo-objects

```
void destroy();
```

Event Service Specification 4

4.1 Service Description

4.1.1 Overview

A standard CORBA request results in the synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated between the client and the server. A request is directed to a particular object. For the request to be successful, both the client and the server must be available. If a request fails because the server is unavailable, the client receives an exception and must take some appropriate action.

In some scenarios, a more decoupled communication model between objects is required. For example:

- A system administration tool is interested in knowing if a disk runs out of space. The software managing a disk is unaware of the existence of the system administration tool. The software simply reports that the disk is full. When a disk runs out of space, the system administration tool opens a window to inform the user which disk has run out of space.
- A property list object is associated with an application object. The property list object is physically separate from the application object. The application object is interested in the changes made to its properties by a user. The properties can be changed without involving the application object. That is, in order to have reasonable response time for the user, changing a property does not activate the application object. However, when the application object is activated, it needs to know about the changes to its properties.
- A CASE tool is interested in being notified when a source program has been modified. The source program simply reports when it is modified. It is unaware of the existence of the CASE tool. In response to the notification, the CASE tool invokes a compiler.

- Several documents are linked to a spreadsheet. The documents are interested in knowing when the value of certain cells have changed. When the cell value changes, the documents update their presentations based on the spreadsheet. Furthermore, if a document is unavailable because of a failure, it is still interested in any changes to the cells and wants to be notified of those changes when it recovers.

4.1.2 Event Communication

The Event Service decouples the communication between objects. The Event Service defines two roles for objects: the supplier role and the consumer role. *Suppliers* produce event data and *consumers* process event data. Event data are communicated between suppliers and consumers by issuing standard CORBA requests.

There are two approaches to initiating event communication between suppliers and consumers, and two orthogonal approaches to the form that the communication can take.

The two approaches to initiating event communication are called the *push model* and the *pull model*. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

The communication itself can be either generic or typed. In the generic case, all communication is by means of generic push or pull operations that take a single parameter that packages all the event data. In the typed case, communication is via operations defined in OMG IDL. Event data is passed by means of the parameters, which can be defined in any manner desired. Section 4.2 through section 4.5 discuss generic event communication in detail; section 4.6 through section 4.9 discuss typed event communication in detail.

An *event channel* is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events. Event channels are standard CORBA objects and communication with an event channel is accomplished using standard CORBA requests.

4.1.3 Example Scenario

This section provides a general scenario that illustrates how the Event Service can be used.

The Event Service can be used to provide “change notification”. When an object is changed (its state is modified), an event can be generated that is propagated to all interested parties. For example, when a spreadsheet cell object is modified, all compound documents which contain a reference (link) to that cell can be notified (so the document can redisplay the referenced cell, or recalculate values that depend on the cell). Similarly, when an engineering specification object is modified, all engineers who have registered an interest in the specification can be notified that the specification has changed.

In this scenario, objects that can be "changed" act as suppliers, parties interested in receiving notifications of changes act as consumers, and one or more event channel objects are used as intermediaries between consumers and suppliers. *Either the push or the pull model can be used at either end.*

If the push model is used by suppliers, objects that can be changed support the *PushSupplier* interface so that event communication can be discontinued, use the *EventChannel*, the *SupplierAdmin* and the *ProxyPushConsumer* interfaces to register as suppliers of events, and use the *ProxyPushConsumer* interface to push events to event channels.

When a change occurs to an object, a changeable object invokes a push operation on the channel. It provides as an argument to the push operation information that describes the event. This information is of data type any - it can be as simple or as complex as is necessary. For example, the event information might identify the object reference of the object that has been changed, it might identify the kind of change that has occurred, it might provide a new displayable image of the changed object or it might identify one or more additional objects that describe the change that has been made.

If the pull model is used by consumers, all client objects that want to be notified of changes support the *PullConsumer* interface so communication can be discontinued, using the *EventChannel*, *ConsumerAdmin* and *ProxyPullSupplier* interfaces to register as consumers of events, and using the *ProxyPullSupplier* interface to pull events from event channels.

The consumer may use either a blocking or non-blocking mechanism for receiving notification of changes. Using the try_pull operation, the consumer can periodically poll the channel for events. Alternatively, the consumer can use the pull operation which will block the consumer's execution thread until an event is generated by some supplier.

Event channels act as the intermediaries between the objects being changed and objects interested in knowing about changes. The channels that provide change notification can be general purpose, well-known objects (e.g., "persistent server-based objects" that are run as part of a workgroup-wide framework of objects that provide "desktop services") or specific-to-task objects (e.g., temporary objects that are created when needed). Objects that use event channels may locate the channels by looking for them in a persistently available server (e.g., by looking for them in a naming service) or they may be given references to these objects as part of a specific-to-task object protocol (e.g., when an "open" operation is invoked on an object, the object may return the reference to an event channel which the caller should use until the object is closed).

Event channels determine how changes are propagated between suppliers and consumers, i.e., the qualities of service (Section 4.1.6). For example, an event channel determines the persistence of an event. The channel may keep an event for a specified period of time, passing it along to any consumer who registers with the channel during that period of time (e.g., it may keep event notifications about changes to engineering specifications for a week). Alternatively, the channel may only pass on events to consumers who are currently waiting for notification of changes (e.g., notifications of changes to a spreadsheet cell may only be sent to consumers who are currently displaying that cell).

This scenario exemplifies one way the event service described here forms a basic building block used in providing higher-level services specific to an application or common facilities framework of objects.

Instead of using the generic event channel, a typed event channel could also have been used.

4.1.4 Design Principles

The Event Service design satisfies the following principles:

- Events work in a distributed environment. The design does not depend on any global, critical, or centralized service.
- Event services allow multiple consumers of an event and multiple event suppliers.
- Consumers can either request events or be notified of events, whichever is more appropriate for application design and performance.
- Consumers and suppliers of events support standard OMG IDL interfaces; no extensions to CORBA are necessary to define these interfaces.
- A supplier can issue a single standard request to communicate event data to all consumers at once.
- Suppliers can generate events without knowing the identities of the consumers. Conversely, consumers can receive events without knowing the identities of the suppliers.
- The Event Service interfaces allow multiple qualities of service, for example, for different levels of reliability. It also allows for future interface extensions, such as for additional functionality.
- The Event Service interfaces are capable of being implemented and used in different operating environments, for example, in environments that support threading and those that do not.

4.1.5 Resolution of Technical Issues

This specification addresses the issues identified for event services in the OMG *Object Services Architecture*[1] document as follows:

- **Distributed environment:** The interfaces are designed to allow consumers and suppliers of events to be disconnected from time to time, and do not require centralized event identification, processing, routing, or other services that might be a bottleneck or a single point of failure.

1. *Object Services Architectur*e, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

Events themselves are *not* objects because the CORBA distributed object model does not support passing objects by value.

Event generation: The specification describes how events are generated and delivered in a very general fashion, with event channels as intermediate routing points. It does not require (or preclude) polling, nor does it require that an event supplier directly notify every interested party.

Events involving multiple objects: Complex events may be handled by constructing a notification tree of event consumer/suppliers checking for successively more specific event predicates. The specification does not require a general or global event predicate evaluation service as this may not be sufficiently reliable, efficient, or secure in a distributed, heterogeneous (potentially decoupled) environment.

Scoping, grouping, and filtering events: The specification takes advantage of CORBA's distributed scoping and grouping mechanisms for the identifier and type of events. Event filtering is easily achieved through event channels that selectively deliver events from suppliers to consumers. Event channels can be composed; that is, one event channel can consume events supplied by another. Typed event channels can provide filtering based on event type.

Registration and generation of events: Consumers and suppliers register with event channels themselves. Event channels are objects and they are found by any fashion that objects can be found. A global registration service is not required; any object that conforms to the IDL interface may consume an event.

Event parameters: The specification supports a parameter of type any that can be delivered with an event, used for application-specific data.

Forgery and secure events: Because event suppliers are objects, the specification leverages any ORB work on security for object references and communication.

Performance: The design is a minimalist one, and requires only one ORB call per event received. It supports both push-style and pull-style notification to avoid inefficient event polling. Since event suppliers, consumers, and channels are all ORB objects, the service directly benefits from a Library Object Adapter or any other ORB optimizations.

Formalized Event Information: For specific application environments and frameworks it may be beneficial to formalize the data associated with an event (defined in this specification as type any). This can be accomplished by defining a typed structure for this information. Depending on the needs of the environment, the kinds of information included might be a priority, timestamp, origin string, and confirmation indicator. This information might be solely for the benefit of the event consumer or might also be interpreted by particular event channel implementations.

Confirmation of Reception: Some applications may require that consumers of an event provide an explicit confirmation of reception back to the supplier. This can be supported effectively using a "reverse" event channel through which consumers send back confirmations as normal events. This obviates the need for any special confirmation mechanism. However, strict atomic delivery between all suppliers and all consumers requires additional interfaces.

4.1.6 Quality of Service

Application domains requiring event-style communication have diverse reliability requirements, from "at-most-once" semantics (best effort) to guaranteed "exactly-once" semantics, availability requirements, throughput requirements, performance requirements (i.e., how fast events are disseminated), and scalability requirements.

Clearly no single implementation of the Event Service can optimize such a diverse range of technical requirements. Hence, multiple implementations of event services are to be expected, with different services targeted toward different environments. As such, the event interfaces do not dictate *qualities of service*. Different implementations of the Event Service interfaces can support different qualities of service to meet different application needs.

For example, an implementation that trades at most once delivery to a single consumer in favor of performance is useful for some applications; an implementation that favors performance but cannot preclude duplicate delivery is useful for other applications. Both are acceptable implementations of the interfaces described in this chapter.

Clearly, an implementation of an event channel that discards all events is *not a useful* implementation. Useful implementations will at least support "best-effort" delivery of events.

Note that the interfaces defined in this chapter are incomplete for implementations that support strict notions of atomicity. That is, additional interfaces are needed by an implementation to guarantee that either all consumers receive an event or none of the consumers receive an event; and that all events are received in the same order by all consumers.

4.2 Generic Event Communication

There are two basic models for communicating event data between suppliers and consumers: the *push model* and the *pull model*.

4.2.1 Push Model

In the push model, suppliers "push" event data to consumers; that is, suppliers communicate event data by invoking push operations on the *PushConsumer* interface.

To set up a push-style communication, consumers and suppliers exchange *PushConsumer* and *PushSupplier* object references. Event communication can be broken by invoking a disconnect_push_consumer operation on the *PushConsumer* interface or by invoking a disconnect_push_supplier operation on the *PushSupplier* interface. If the *PushSupplier* object reference is nil, the connection cannot be broken via the supplier.

Figure 4-1 illustrates push-style communication between a supplier and a consumer.

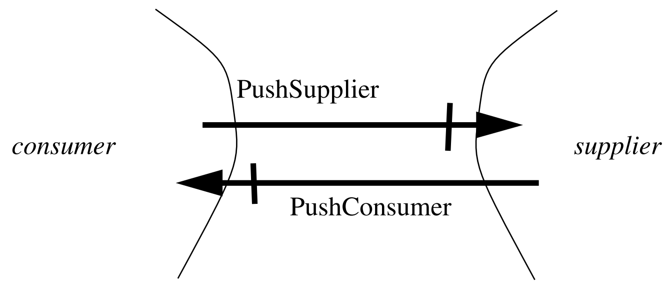


Figure 4-1 Push-style Communication Between a Supplier and a Consumer

4.2.2 *Pull Model*

In the pull model, consumers "pull" event data from suppliers; that is, consumers request event data by invoking pull operations on the *PullSupplier* interface.

To set up a pull-style communication, consumers and suppliers must exchange *PullConsumer* and *PullSupplier* object references. Event communication can be broken by invoking a disconnect_pull_consumer operation on the *PullConsumer* interface or by invoking a disconnect_pull_supplier operation on the *PullSupplier* interface. If the *PullConsumer* object reference is nil, the connection cannot be broken via the consumer.

Figure 4-2 illustrates pull-style communication between a supplier and a consumer.

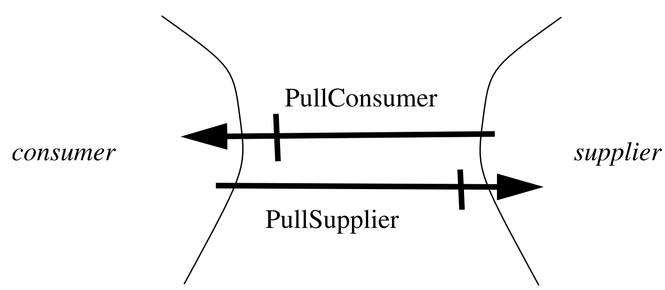


Figure 4-2 Pull-style Communication Between a Supplier and a Consumer

4.3 The CosEventComm Module

The communication styles shown in Figure 4-1 and Figure 4-2 are both supported by four simple interfaces: *PushConsumer*, *PushSupplier*, and *PullSupplier* and *PullConsumer*. These interfaces are defined in an OMG IDL module named CosEventComm, as shown in Figure 4-3.

```
module CosEventComm {

        exception Disconnected{};

        interface PushConsumer {
                void push (in any data) raises(Disconnected);
                void disconnect_push_consumer();
        };

    interface PushSupplier {
                void disconnect_push_supplier();
        };

        interface PullSupplier {
                any pull () raises(Disconnected);
                any try_pull (out boolean has_event)
                        raises(Disconnected);
                void   disconnect_pull_supplier();
        };

        interface PullConsumer {
                void disconnect_pull_consumer();
        };

};
```

Figure 4-3 The OMG IDL Module CosEventComm

4.3.1 The PushConsumer Interface

A push-style consumer supports the *PushConsumer* interface to receive event data.

```
        interface PushConsumer {
                void push (in any data) raises(Disconnected);
                void disconnect_push_consumer();
        };
```

A supplier communicates event data to the consumer by invoking the push operation and passing the event data as a parameter. If the event communication has already been disconnected, the Disconnected exception is raised.

The disconnect_push_consumer operation terminates the event communication; it releases resources used at the consumer to support the event communication. The *PushConsumer* object reference is disposed.

4.3.2 The PushSupplier Interface

A push-style supplier supports the *PushSupplier* interface.

```
interface PushSupplier {
        void disconnect_push_supplier();
};
```

The disconnect_push_supplier operation terminates the event communication; it releases resources used at the supplier to support the event communication. The *PushSupplier* object reference is disposed.

4.3.3 The PullSupplier Interface

A pull-style supplier supports the *PullSupplier* interface to transmit event data.

```
interface PullSupplier {
        any pull () raises(Disconnected);
        any try_pull (out boolean has_event)
                raises(Disconnected);
        void   disconnect_pull_supplier();
};
```

A consumer requests event data from the supplier by invoking either the pull operation or the try_pull operation on the supplier.

- The pull operation blocks until the event data is available or an exception is raised.[2] It returns the event data to the consumer. If the event communication has already been disconnected, the Disconnected exception is raised.
- The try_pull operation does not block: if the event data is available, it returns the event data and sets the has_event parameter to *true*; if the event is not available, it sets the has_event parameter to *false* and the event data is returned as long with an undefined value. If the event communication has already been disconnected, the Disconnected exception is raised.

2. This, of course, may be a standard CORBA exception.

The disconnect_pull_supplier operation terminates the event communication; it releases resources used at the supplier to support the event communication. The *PullSupplier* object reference is disposed.

4.3.4 The PullConsumer Interface

A pull-style consumer supports the *PullConsumer* interface.

```
interface PullConsumer {
        void disconnect_pull_consumer();
};
```

The disconnect_pull_consumer operation terminates the event communication; it releases resources used at the consumer to support the event communication. The *PullConsumer* object reference is disposed.

4.4 Event Channels

The *event channel* is a service that decouples the communication between suppliers and consumers. The event channel is itself both a consumer and a supplier of the event data.

An event channel can provide asynchronous communication of event data between suppliers and consumers. Although consumers and suppliers communicate with the event channel using standard CORBA requests, the event channel does not need to supply the event data to its consumer at the same time it consumes the data from its supplier.

4.4.1 Push-Style Communication with an Event Channel

The supplier pushes event data to the event channel; the event channel, in turn, pushes event data to the consumer. Figure 4-4 illustrates a push-style communication between a supplier and the event channel, and a consumer and the event channel.

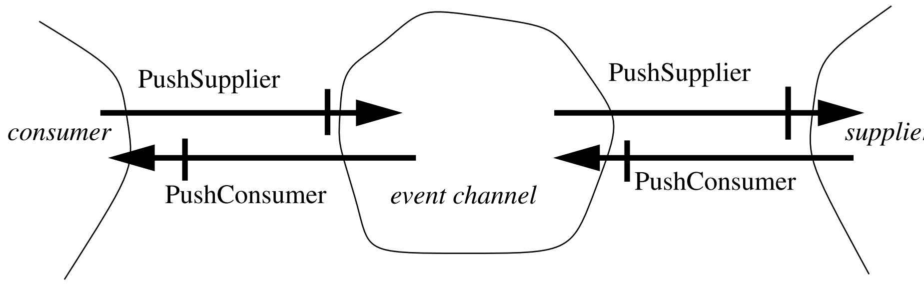


Figure 4-4 Push-style Communication Between a Supplier and an Event Channel, and a Consumer and an Event Channel

4.4.2 Pull-Style Communication with an Event Channel

The consumer pulls event data from the event channel; the event channel, in turn, pulls event data from the supplier. Figure 4-5 illustrates a pull-style communication between a supplier and the event channel, and a consumer and the event channel.

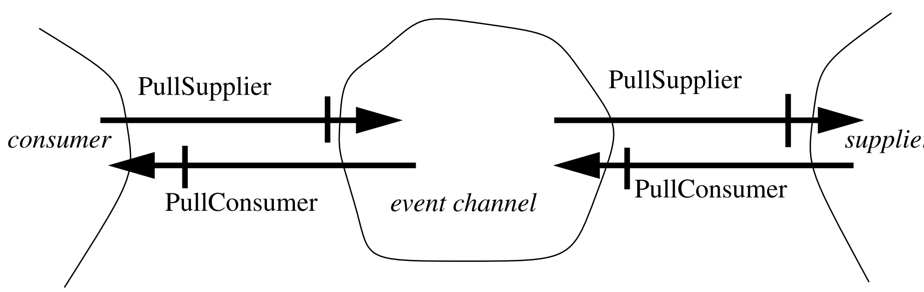


Figure 4-5 Pull-style communication between a supplier and an event channel and a consumer and the event channel

4.4.3 Mixed Style Communication with an Event Channel

An event channel can communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication.

Figure 4-6 illustrates a push-style communication between a supplier and an event channel, and a pull-style communication between a consumer and the event channel. The consumer pulls the event data that the supplier has pushed to the event channel.

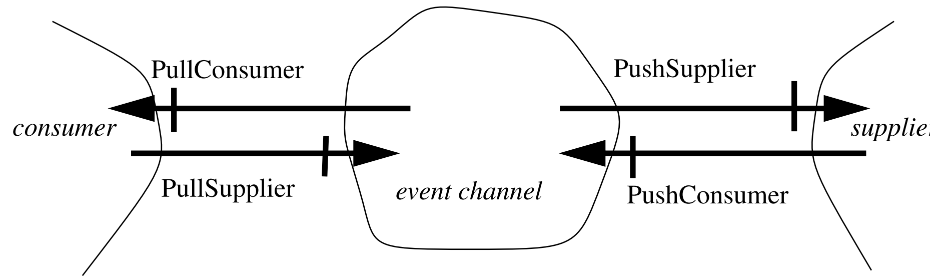


Figure 4-6 Push-style Communication Between a Supplier and an Event Channel, and Pull-style Communication Between a Consumer and an Event Channel

4.4.4 Multiple Consumers and Multiple Suppliers

Figure 4-4, Figure 4-5, and Figure 4-6 illustrate event channels with a single supplier and a single consumer. An event channel can also provide many-to-many communication. The channel consumes events from one or more suppliers, and supplies events to one or more consumers. Subject to the quality of service of a particular implementation, an event channel provides an event to all consumers.

Figure 4-7 illustrates an event channel with multiple push-style consumers and multiple push-style suppliers.

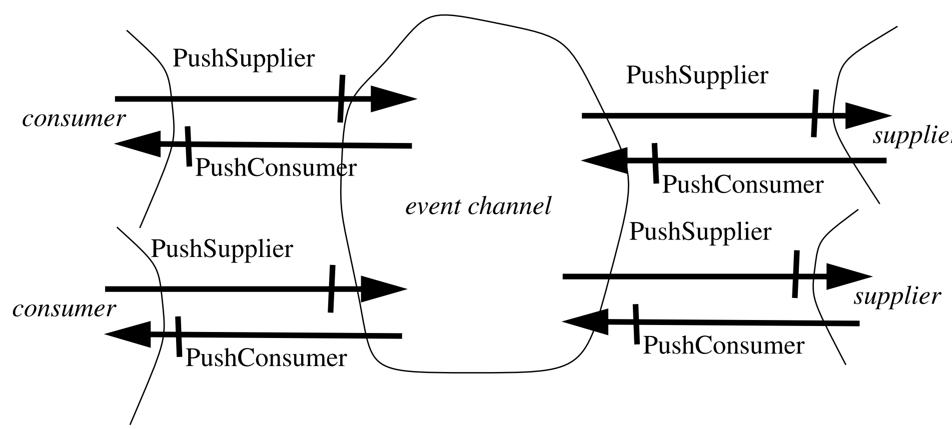


Figure 4-7 An Event Channel with Multiple Suppliers and Multiple Consumers

An event channel can support consumers and suppliers using different communication models.

If an event channel has at least one push-style consumer or at least one pending pull request, the event channel requires an event. If the event channel has pull suppliers, it will issue a request on a pull supplier to satisfy its requirement.

4.4.5 Event Channel Administration

The event channel is built up incrementally. When an event channel is created, no suppliers or consumers are connected to the event channel. Upon creation of the channel, the factory returns an object reference that supports the *EventChannel* interface, as illustrated in Figure 4-8.

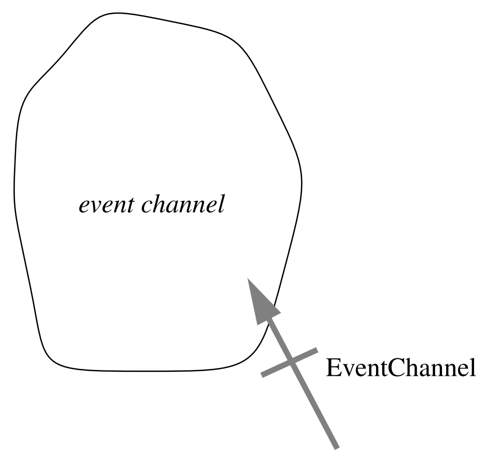


Figure 4-8 A newly created event channel. The channel has no suppliers or consumers.

The *EventChannel* interface defines three administrative operations: an operation returning a *ConsumerAdmin* object for adding consumers, an operation returning a *SupplierAdmin* object for adding suppliers, and an operation for destroying the channel.

The operations for adding consumers return *proxy suppliers*. A proxy supplier is similar to a normal supplier (in fact, it inherits the interface of a supplier), but includes an additional method for connecting a consumer to the proxy supplier.

The operations for adding suppliers return *proxy consumers*. A proxy consumer is similar to a normal consumer (in fact, it inherits the interface of a consumer), but includes an additional method for connecting a supplier to the proxy consumer.

Registration of a producer or consumer is a two step process. An event-generating application first obtains a proxy consumer from a channel, then “connects” to the proxy consumer by providing it with a supplier. Similarly, an event-receiving application first obtains a proxy supplier from a channel, then “connects” to the proxy supplier by providing it with a consumer.

The reason for the two-step registration process is to support composing event channels by an external agent. Such an agent would compose two channels by obtaining a proxy supplier from one and a proxy consumer from the other, and passing each of them a reference to the other as part of their connect operation.

Proxies are in one of three states: *disconnected*, *connected* or *destroyed*. Figure 4-9 gives a state diagram for a proxy. The nodes of the diagram are the states and the edges are labelled with the operations that change the state of the proxy. Push/pull operations are only valid in the *connected* state.

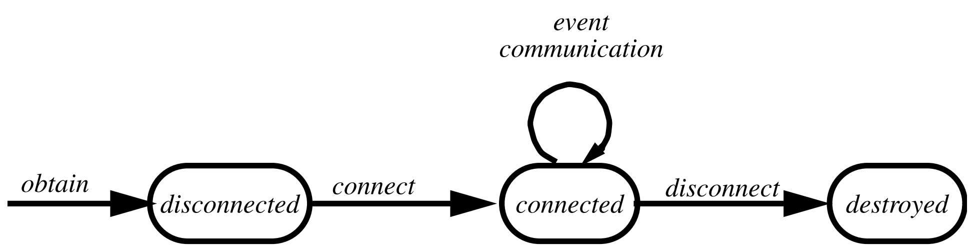


Figure 4-9 State diagram of a proxy.

4.5 The CosEventChannelAdmin Module

The CosEventChannelAdmin module defines the interfaces for making connections between suppliers and consumers. The CosEventChannelAdmin module is defined in Figure 4-10.

```
#include "CosEventComm.idl"

module CosEventChannelAdmin {

	exception AlreadyConnected {};
	exception TypeError {};

	interface ProxyPushConsumer: CosEventComm::PushConsumer {
		void connect_push_supplier(
				in CosEventComm::PushSupplier push_supplier)
			raises(AlreadyConnected);
	};

	interface ProxyPullSupplier: CosEventComm::PullSupplier {
		void connect_pull_consumer(
				in CosEventComm::PullConsumer pull_consumer)
			raises(AlreadyConnected);
	};

	interface ProxyPullConsumer: CosEventComm::PullConsumer {
		void connect_pull_supplier(
				in CosEventComm::PullSupplier pull_supplier)
			raises(AlreadyConnected,TypeError);
	};

	interface ProxyPushSupplier: CosEventComm::PushSupplier {
		void connect_push_consumer(
				in CosEventComm::PushConsumer push_consumer)
			raises(AlreadyConnected, TypeError);
	};
```

```
    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
    };

    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
    };

    interface EventChannel {
        ConsumerAdmin for_consumers();
        SupplierAdmin for_suppliers();
        void destroy();
    };

};
```

Figure 4-10 The CosEventChannelAdmin Module

4.5.1 The EventChannel Interface

The *EventChannel* interface defines three administrative operations: adding consumers, adding suppliers, and destroying the channel.

```
interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
};
```

Any object that possesses an object reference that supports the *EventChannel* interface can perform these operations:

- The *ConsumerAdmin* interface allows consumers to be connected to the event channel. The for_consumers operation returns an object reference that supports the *ConsumerAdmin* interface.
- The *SupplierAdmin* interface allows suppliers to be connected to the event channel. The for_suppliers operation returns an object reference that supports the *SupplierAdmin* interface.
- The destroy operation destroys the event channel.

Consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers. For example, a creator might wish to be the sole supplier of event data but allow many consumers to be connected to the channel. In such a case, the creator would simply export the *ConsumerAdmin* object.

4.5.2 The ConsumerAdmin Interface

The *ConsumerAdmin* interface defines the first step for connecting consumers to the event channel; clients use it to obtain proxy suppliers.

```
interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
};
```

The obtain_push_supplier operation returns a *ProxyPushSupplier* object. The *ProxyPushSupplier* object is then used to connect a push-style consumer.

The obtain_pull_supplier operation returns a *ProxyPullSupplier* object. The *ProxyPullSupplier* object is then used to connect a pull-style consumer.

4.5.3 The SupplierAdmin Interface

The *SupplierAdmin* interface defines the first step for connecting suppliers to the event channel; clients use it to obtain proxy consumers.

```
interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
};
```

The obtain_push_consumer operation returns a *ProxyPushConsumer* object. The *ProxyPushConsumer* object is then used to connect a push-style supplier.

The obtain_pull_consumer operation returns a *ProxyPullConsumer* object. The *ProxyPullConsumer* object is then used to connect a pull-style supplier.

4.5.4 The ProxyPushConsumer Interface

The *ProxyPushConsumer* interface defines the second step for connecting push suppliers to the event channel.

```
interface ProxyPushConsumer: CosEventComm::PushConsumer {
        void connect_push_supplier(
                        in CosEventComm::PushSupplier push_supplier)
                raises(AlreadyConnected);
};
```

A nil object reference may be passed to the connect_push_supplier operation; if so a channel cannot invoke the disconnect_push_supplier operation on the supplier; the supplier may be disconnected from the channel without being informed.

If the *ProxyPushConsumer* is already connected to a *PushSupplier*, then the AlreadyConnected exception is raised.

4.5.5 The ProxyPullSupplier Interface

The *ProxyPullSupplier* interface defines the second step for connecting pull consumers to the event channel.

```
interface ProxyPullSupplier: CosEventComm::PullSupplier {
	void connect_pull_consumer(
			in CosEventComm::PullConsumer pull_consumer)
		raises(AlreadyConnected);
};
```

A nil object reference may be passed to the connect_pull_consumer operation; if so a channel cannot invoke a disconnect_pull_consumer operation on the consumer; the consumer may be disconnected from the channel without being informed.

If the *ProxyPullSupplier* is already connected to a *PullConsumer*, then the AlreadyConnected exception is raised.

4.5.6 The ProxyPullConsumer Interface

The *ProxyPullConsumer* interface defines the second step for connecting pull suppliers to the event channel.

```
interface ProxyPullConsumer: CosEventComm::PullConsumer {
	void connect_pull_supplier(
			in CosEventComm::PullSupplier pull_supplier)
		raises(AlreadyConnected, TypeError);
};
```

Implementations should raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the connect_pull_supplier operation.

If the *ProxyPullConsumer* is already connected to a *PullSupplier*, then the AlreadyConnected exception is raised.

An implementation of a *ProxyPullConsumer* may put additional requirements on the interface supported by the pull supplier. If the pull supplier does not meet those requirements the *ProxyPullConsumer* raises the TypeError exception. (See section 4.7.2 for an example.)

4.5.7 The ProxyPushSupplier Interface

The *ProxyPushSupplier* interface defines the second step for connecting push consumers to the event channel.

```
interface ProxyPushSupplier: CosEventComm::PushSupplier {
    void connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer)
        raises(AlreadyConnected, TypeError);
};
```

Implementations should raise the CORBA standard BAD_PARAM exception if a nil object reference is passed to the connect_push_consumer operation.

If the *ProxyPushSupplier* is already connected to a *PushConsumer*, then the AlreadyConnected exception is raised.

An implementation of a *ProxyPushSupplier* may put additional requirements on the interface supported by the push consumer. If the push consumer does not meet those requirements the *ProxyPushSupplier* raises the TypeError exception. (See section 4.7.1 for an example.)

4.6 Typed Event Communication

Section 4.2 discusses generic event communication using push and pull operations. The next few sections describe how event communication can be described in OMG IDL and how typed event channels can support such typed event communication.

4.6.1 Typed Push Model

In the typed push model, suppliers call operations on consumers using some mutually agreed interface *I*. The interface *I* is defined in IDL, and may contain any operations subject to the following restrictions:

- All parameters must be in parameters only.
- No return values are permitted

These are the same restrictions as CORBA imposes on oneway operations, and for similar reasons: event communication is unidirectional, and does not directly support responses. The operations can be declared oneway, but need not be.

To set up typed push-style communication, consumers and suppliers exchange *TypedPushConsumer* and *PushSupplier* object references. (Note that the supplier interface is the same as the untyped case.) The supplier then invokes the get_typed_consumer operation of the *TypedPushConsumer* interface, which returns an object reference supporting the typed interface, *I*, referred to as an *I-reference*. The particular interface, *I*, that the reference supports is dependent on the particular

TypedPushConsumer, and must be mutually agreed by supplier and consumer. Once the supplier has obtained the *I*-reference, it can call operations in interface *I* on the consumer.

As in the case of the generic push-style, event communication can be broken by invoking a disconnect_push_consumer operation on the *TypedPushConsumer* interface or by invoking a disconnect_push_supplier operation on the *PushSupplier* interface. If the *PushSupplier* object reference is nil, the connection cannot be broken via the supplier.

Figure 4-11 illustrates typed push-style communication between supplier and consumer.

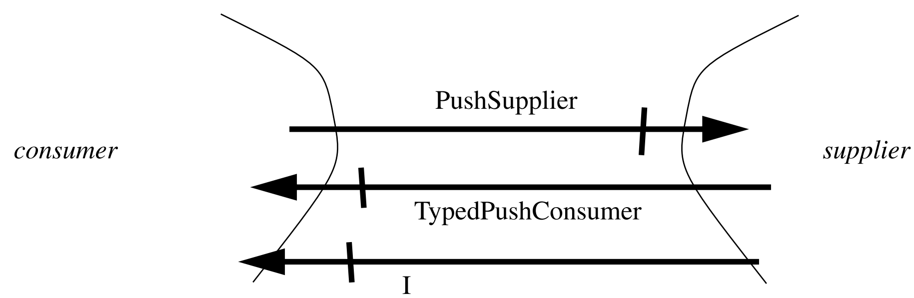


Figure 4-11 Typed Push-style Communication Between a Supplier and a Consumer

4.6.2 Typed Pull Model

In the typed pull model, consumers call operations on suppliers, requesting event information, using some mutually agreed interface *Pull<I>*[3]. For every interface *I* having the properties described in section 4.6.1, an interface *Pull<I>* is defined as follows:

- For every operation o in *I, Pull<I>* contains two operations:
  - pull_o, with all in parameters changed to out parameters. When called, this operation will return with the event data in the *out* parameters. If no *o*-event is currently available, it will block.
  - boolean try_o, with all in parameters changed to out parameters. When called, this operation will check whether an *o*-event is currently available. If so, it will return true, with the event data in the *out* parameters. If not, it will return false, with the out parameters undefined

3. *Pull<I>* is used as notation for a computed interface from interface I. Thus, if *I* is an interface *DocumentEvents*, *Pull<I>* is an interface *PullDocumentEvents.*

The interface *Pull<I>* is designed to allow pulling of exactly the same events that can be pushed using interface *I*.

To set up typed pull-style communication, consumers and suppliers exchange *PullConsumer* and *TypedPullSupplier* object references. (Note that the consumer interface is the same as the untyped case.) The consumer then invokes the get_typed_supplier operation of the *TypedPullSupplier*, which returns an object reference supporting the typed interface, *Pull<I>*, referred to as a *Pull<I>-reference*. The particular interface, *Pull<I>*, that the reference supports is dependent on the particular *TypedPullSupplier*, and must be mutually agreed by supplier and consumer. Once the consumer has obtained the *Pull<I>-reference*, it can call operations in interface *Pull<I>* on the supplier.

Figure 4-12 illustrates typed pull-style communication between supplier and consumer.

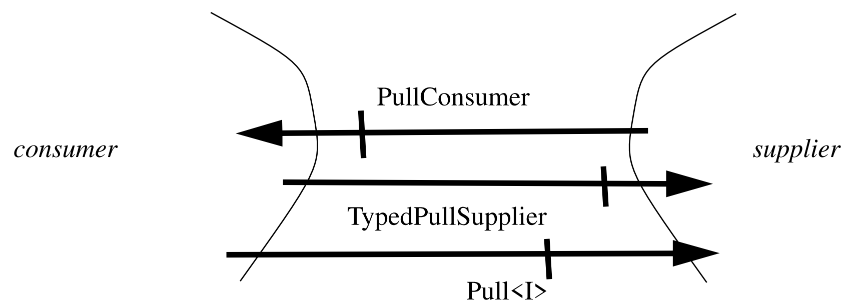


Figure 4-12 Typed Pull-style Communication Between a Supplier and a Consumer

4.7 The CosTypedEventComm Module

The typed communication styles shown in Figure 4-11 and Figure 4-12 are both supported by two new interfaces, *TypedPushConsumer* and *TypedPullSupplier* and two existing interfaces, *PushSupplier* and *PullConsumer*. The first two interfaces are defined in an OMG IDL module named *CosTypedEventComm*, as shown in Figure 4-13. The last two are the same as for untyped event communication, and were defined in the *CosEventComm* module in Figure 4-3.

```
#include "CosEventComm.idl"

module CosTypedEventComm {

    interface TypedPushConsumer : CosEventComm::PushConsumer {
        Object get_typed_consumer();
    };

    interface TypedPullSupplier : CosEventComm::PullSupplier {
        Object get_typed_supplier();
    };

};
```

Figure 4-13 The IDL Module CosTypedEventComm

4.7.1 The TypedPushConsumer Interface

A typed push-style consumer supports the *TypedPushConsumer* interface both to receive event data in the generic manner, and to supply a specific typed interface through which to receive it in typed form.

```
    interface TypedPushConsumer : CosEventComm::PushConsumer {
        Object get_typed_consumer();
    };
```

The *TypedPushConsumer* can behave just like an untyped *PushConsumer*, described in section 4.3.1. In addition, if the supplier wishes to communicate event data to the consumer in typed rather than generic form, it first invokes the get_typed_consumer operation. This returns an *I-reference* supporting an interface *I*. The particular interface, *I*, that the reference supports is dependent on the particular *TypedPushConsumer*. The return type of the operation is *Object*, because different *TypedPushConsumers* will return references of different types, so the actual type cannot be specified in a general definition. Once the supplier has obtained the *I-reference*, it can narrow it to *I*, and then call operations in interface *I* on the consumer. Mutual agreement about *I* is needed between the supplier and consumer. If they do not agree, the narrow operation will fail.

As noted above, a *TypedPushConsumer* must support the push operation, inherited from Cos*EventComm::PushConsumer*. Implementing push fully is an unnecessary burden if the consumer is intended for typed use only. It is therefore permissible to implement a *TypedPushConsumer* with a null implementation of push that merely raises the standard CORBA exception NO_IMPLEMENT. Clearly, suppliers must know this and confine themselves to typed communication with such consumers.

4.7.2 The TypedPullSupplier Interface

A typed pull-style supplier supports the *TypedPullSupplier* interface both to allow consumers to pull event data in the generic manner, and to supply a specific typed interface through which they can pull it in typed form.

```
interface TypedPullSupplier : CosEventComm::PullSupplier {
        Object get_typed_supplier();
};
```

The *TypedPullSupplier* can behave just like an untyped *PullSupplier*, described in section 4.3.3. In addition, if the consumer wishes to pull event data from the supplier in typed rather than generic form, it first invokes the get_typed_supplier operation. This returns a *Pull<I>-reference* supporting an interface *Pull<I>*. The particular interface, *Pull<I>*, that the reference supports is dependent on the particular *TypedPullSupplier*. The return type of the operation is *Object*, because different *TypedPullSuppliers* will return references of different types, so the actual type cannot be specified in a general definition. Once the consumer has obtained the *Pull<I>-reference*, it can narrow it to *Pull<I>*, and then call operations in interface *Pull<I>* on the supplier. Mutual agreement about *Pull<I>* is needed between the supplier and consumer. If they do not agree, the narrow operation will fail.

As noted above, a *TypedPullSupplier* must support the pull and try_pull operations, inherited from Cos*EventComm::PullSupplier*. Implementing these operations fully is an unnecessary burden if the supplier is intended for typed use only. It is therefore permissible to implement a *TypedPullSupplier* with null implementations of pull and try_pull that merely raise the standard CORBA exception NO_IMPLEMENT. Clearly, consumers must know this and confine themselves to typed communication with such suppliers.

4.8 Typed Event Channels

Typed event channels are analogous to generic event channels, but they support both typed and generic event communication. These forms can be mixed at will. A single channel can handle events supplied and consumed in any combination of the forms defined earlier (push/pull, generic/typed). An event supplied in typed form can be consumed in generic form, or vice versa.[4]

4. Doing this does require an understanding on the part of the generic suppliers and consumers of how the channel packages parameters of typed calls when converting them to generic form. Details of this packaging are dependent on the implementation of the channel.

4.9 The CosTypedEventChannelAdmin Module

The CosTypedEventChannelAdmin module defines the interfaces for making connections between suppliers and consumers that use either generic or typed communication. It is defined in Figure 4-14. Most of its interfaces are specializations of the corresponding interfaces in the CosEventChannel module defined in Figure 4-10.

```
#include "CosEventChannel.idl"
#include "CosTypedEventComm.idl"

module CosTypedEventChannelAdmin {

        exception InterfaceNotSupported {};
        exception NoSuchImplementation {};
        typedef string Key;

        interface TypedProxyPushConsumer :
                        CosEventChannelAdmin::ProxyPushConsumer,
                        CosTypedEventComm::TypedPushConsumer  { };

        interface TypedProxyPullSupplier :
                        CosEventChannelAdmin::ProxyPullSupplier,
                        CosTypedEventComm::TypedPullSupplier { };

        interface TypedSupplierAdmin :
                        CosEventChannelAdmin::SupplierAdmin {
                TypedProxyPushConsumer obtain_typed_push_consumer(
                                in Key supported_interface)
                        raises(InterfaceNotSupported);
                ProxyPullConsumer obtain_typed_pull_consumer (
                                in Key uses_interface)
                         raises(NoSuchImplementation);
        };

        interface TypedConsumerAdmin :
                        CosEventChannelAdmin::ConsumerAdmin {
                TypedProxyPullSupplier obtain_typed_pull_supplier(
                                in Key supported_interface)
                        raises (InterfaceNotSupported);
                ProxyPushSupplier obtain_typed_push_supplier(
                                in Key uses_interface)
                        raises(NoSuchImplementation);
        };

        interface TypedEventChannel {
                TypedConsumerAdmin for_consumers();
                TypedSupplierAdmin for_suppliers();
                void destroy ();
        };
};
```

Figure 4-14 The CosTypedEventChannelAdmin Module

4.9.1 The TypedEventChannel Interface

```
interface TypedEventChannel {
	TypedConsumerAdmin for_consumers();
	TypedSupplierAdmin for_suppliers();
	void destroy ();
};
```

This interface is analogous to CosEventChannelAdmin::EventChannel. However, it returns typed versions of the consumer and supplier administration interfaces, which are capable of providing proxies for either generic or typed communication.

4.9.2 The TypedConsumerAdmin Interface

The *TypedConsumerAdmin* interface defines the first step for connecting consumers to typed event channel; clients use it to obtain proxy suppliers.

```
interface TypedConsumerAdmin :
		CosEventChannelAdmin::ConsumerAdmin {
	TypedProxyPullSupplier obtain_typed_pull_supplier(
			in Key supported_interface)
		raises (InterfaceNotSupported);
	ProxyPushSupplier obtain_typed_push_supplier(
			in Key uses_interface)
		raises(NoSuchImplementation);
};
```

The obtain_typed_pull_supplier operation takes a Key parameter that identifies an interface, *Pull<I>*. The scope of the key is the typed event channel. It returns a *TypedProxyPullSupplier* for interface *Pull<I>*. The *TypedProxyPullSupplier* will allow an attached pull consumer to pull events either in generic form or using operations in interface *Pull<I>*. It is up to the implementation of obtain_typed_pull_supplier to create or find an appropriate *TypedProxyPullSupplier.* If it cannot, it raises the exception InterfaceNotSupported.

The obtain_typed_push_supplier operation takes a Key parameter that identifies an interface, *I.* The scope of the key is the typed event channel. It returns a *ProxyPushSupplier* that calls operations in interface *I*, rather than push operations. It is up to the implementation of obtain_typed_push_supplier to create or find an appropriate *ProxyPushSupplier*[5]. If it cannot, it raises the exception NoSuchImplementation.

5. see Appendix A for implementation considerations.

Such a *ProxyPushSupplier* is guaranteed only to invoke operations defined in interface *I*. Any event on the channel that does not correspond to an operation defined in interface *I* is not passed on to the consumer. Such a *ProxyPushSupplier* is therefore an event filter based on type.

4.9.3 The TypedSupplierAdmin Interface

The *TypedSupplierAdmin* interface defines the first step for connecting suppliers to the typed event channel; clients use it to obtain proxy consumers.

```
interface TypedSupplierAdmin :
            CosEventChannelAdmin::SupplierAdmin {
        TypedProxyPushConsumer obtain_typed_push_consumer(
                    in Key supported_interface)
            raises(InterfaceNotSupported);
        ProxyPullConsumer obtain_typed_pull_consumer (
                    in Key uses_interface)
            raises(NoSuchImplementation);
};
```

The obtain_typed_push_consumer operation takes a Key parameter that identifies an interface, *I*. The scope of the key is the typed event channel. It returns a *TypedProxyPushConsumer* for *I*. An attached supplier can provide events by using operations in interface *I*. It is up to the implementation of obtain_typed_push_consumer to create or find an appropriate *TypedProxyPushConsumer*. If it cannot, it raises the exception InterfaceNotSupported.

The obtain_typed_pull_consumer operation takes a Key parameter that identifies an interface, *Pull<I>*. The scope of the key is the typed event channel. It returns a *ProxyPullConsumer* that calls operations in interface *Pull<I>*, rather than pull operations. It is up to the implementation of obtain_typed_pull_consumer to create or find an appropriate *ProxyPullConsumer*. If it cannot, it raises the exception NoSuchImplementation.

Such a *ProxyPullConsumer* is guaranteed only to invoke operations defined in interface *Pull<I>*. Any event request that does not correspond to an operation defined in interface *Pull<I>* is not pulled from the supplier. Such a *ProxyPullConsumer* is therefore an event filter based on type.

4.9.4 The TypedProxyPushConsumer Interface

The *TypedProxyPushConsumer* interface defines the second step for connecting push suppliers to the typed event channel.

```
interface TypedProxyPushConsumer :
            CosEventChannelAdmin::ProxyPushConsumer,
            CosTypedEventComm::TypedPushConsumer  { };
```

- By inheriting from both CosEventChannelAdmin::ProxyPushConsumer and CosTypedEventComm::TypedPushConsumer, this interface supports:
- Connection and disconnection of push suppliers, exactly as in the generic event channel,
- Generic push operation and
- Obtaining the typed view, so that the supplier can use typed push communication. The reference returned by get_typed_consumer has the interface identified by the Key used when this *TypedProxyPushConsumer* was obtained. (See section 4.9.3)

4.9.5 The TypedProxyPullSupplier Interface

The *TypedProxyPullSupplier* interface defines the second step for connecting pull consumers to the typed event channel.

```
interface TypedProxyPullSupplier :
        CosEventChannelAdmin::ProxyPullSupplier,
        CosTypedEventComm::TypedPullSupplier { };
```

By inheriting from both CosEventChannelAdmin::ProxyPullSupplier and CosTypedEventComm::TypedPullSupplier, this interface supports:

- Connection and disconnection of pull consumers, exactly as in the generic event channel,
- Generic pull and try_pull operations and
- Obtaining the typed view, so that the consumer can use typed pull communication. The reference returned by get_typed_supplier supports the interface identified by the Key used when this *TypedProxyPullSupplier* was obtained. (See section 4.9.2).

4.10 Composing Event Channels and Filtering

The event channel administration operations defined in section 4.5 support the composition of event channels. That is, one event channel can consume events supplied by another. This architecture allows the implementation of an event channel that filters the events supplied by another.

Since the *ProxyPushSupplier* for interface *I* of a typed event channel only pushes events that correspond to *I*, it acts as a filter based on type. Similarly, the *ProxyPullConsumer* for interface *Pull<I>* of a typed event channel only pulls events that correspond to *Pull<I>*, it also acts as a filter based on type.

4.11 Policies for Finding Event Channels

The Event Service does not establish a policy for finding event channels. Finding a service is orthogonal to using the service. Higher levels of software (such as the desktop) can make policies for using the event channel. That is, higher layers will dictate when an event channel is created and how references to the event channel are obtained. By representing the event channel as an object, it has all of the properties that apply to objects, including support by finding mechanisms.

For example, when a user performs a drag-and-drop or cut-and-paste operation, an event channel could be created and identified to suppliers and consumers. Alternatively, the event channel could be named in a naming context, or it could be exported through an operation on an object.

Appendix A Implementing Typed Event Channels

Note – Implementation details do not form part of an OMG specification, and should not be standardized. On the other hand, it is not obvious that typed channels can be implemented without extensions to CORBA. This section indicates *one* strategy for implementing typed event channels. It is included to show that typed event channels can be implemented; it is not intended in any way to constrain implementations. Optimized implementations are certainly possible.

Figure 4-15 demonstrates a possible implementation of a typed event channel. This appendix concentrates on push style communication. The implementation of pull-style communication is analogous.

The implementation interposes an *encoder* between typed-style suppliers and the channel and a *decoder* between the channel and typed-style consumers.

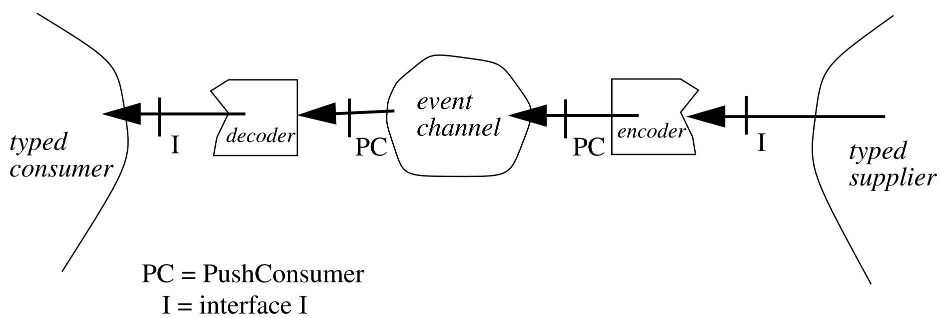


Figure 4-15 A possible implementation of a typed event channel.

At the supplier end, an *encoder* converts operation calls to push calls.

At the consumer end, a *decoder* converts push calls back to operation calls.

The effect of such a communication is thus that the original operation is eventually called on the consumer, but the communication is routed via the channel. Of course, there can be multiple suppliers and multiple consumers on the same channel. Whenever one of the suppliers calls an operation, it is delivered by the channel to all consumers.

The encoder must package the operation identification and the parameters in a manner that the decoder can unpack them correctly.

Given the OMG IDL definition of an interface, *I*, an encoder generator could generate an implementation that supports the interface *I* and converts all calls on this interface to push calls on an event channel.

Similarly, it is possible to generate an I-decoder from the OMG IDL definition of I.

The typed event channel is responsible for finding, creating or implementing the appropriate encoders. An appropriate encoder is found or created in response to the obtain_typed_push_consumer request on the typed event channel. The encoder is returned in response to the get_typed_consumer request.

Similarly, the typed event channel is responsible for finding, creating or implementing the appropriate decoders. An appropriate decoder is found or created in response to the connect_push_consumer request on the typed event channel.

Appendix B An Event Channel Use Example

This section illustrates an example use of the event channel, including the following:

- Creating an event channel
- Consumers and/or suppliers finding the channel
- Suppliers using the event channel
- In this example, the document object creates event channels and defines operations in its interface to allow consumers to be added.
- The *Document* interface defines two operations to return event channels:

```
interface Document {
    ConsumerAdmin title_changed();
    ConsumerAdmin new_section();
        :
};
```

The title_changed operation causes the document to generate an event when its title is changed; the new_section operation causes the document to generate an event when a new section is added. Both operations return *ConsumerAdmin* object references. This allows consumers to be added to the event channel.

- The title_changed implementation contains instance variables for using and administering the event channels.

```
/* Factory for creating event channels. */
EventChannelFactoryRef    ecf;

/* For title changed event channel */
EventChannelRef                          event_channel;

ConsumerAdminRef                     consum_admin;
SupplierAdminRef        supplier_admin;

ProxyPushConsumerRef      proxy_push_consumer;
PushSupplierRef          doc_side_connection;
```

- At some point, the document implementation creates the event channel, gets supplier and consumer administrative references, and adds itself as a supplier[6].

```
event_channel = ecf->create_eventchannel(env);

supplier_admin = event_channel->for_suppliers(env);
consumer_admin = event_channel->for_consumers(env);
proxy_push_consumer = supplier_admin->obtain_push_consumer(env);

proxy_push_consumer->connect_push_supplier(env,
                                           doc_side_connection)
```

- The title_changed operation returns the *ConsumerAdmin* object reference.

```
return consumer_admin;
```

Clients of this operation can add consumers.

- When the title changes, the document implementation pushes the event to the channel.

```
proxy_push_consumer->push(env,data);
```

The document implementation similarly initializes, exports, and uses the event channel for reporting new sections.

6. For readability, exception handling is omitted from these code fragments.

Persistent Object Service Specification 5

5.1 Introduction

The goal of the Persistent Object Service (POS) is to provide common interfaces to the mechanisms used for retaining and managing the persistent state of objects. The Persistent Object Service will be used in conjunction with other object services, for example, naming, relationships, transactions, life cycle, and so forth. The Persistent Object Service has the primary responsibility for storing the persistent state of objects, with other services providing other capabilities.

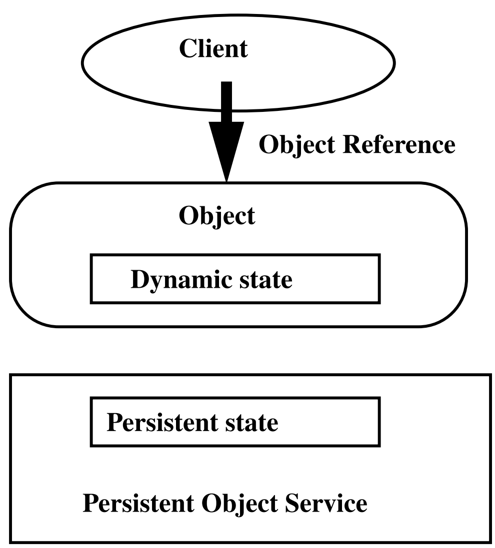


Figure 5-1 Roles in the Persistent Object Service

Figure 5-1 shows the participants in the Persistent Object Service. The state of the object can be considered in two parts, the *dynamic state*, which is typically in memory and is not likely to exist for the whole lifetime of the object (for example, it would not be preserved in the event of a system failure), and the *persistent state*, which the object could use to reconstruct the dynamic state.

Although the ORB provides the ability for an object reference to be persistent, it cannot ensure that the state of the object will be available just because the object reference is still valid.

The object ultimately has the responsibility of managing its state, but can use or delegate to the Persistent Object Service for the actual work. There is no requirement that any object use any particular persistence mechanism. For example, it may write its data to files using non-CORBA interfaces, or a single-level-store mechanism may be used. However, the Persistent Object Service provides capabilities that should be useful to a wide variety of objects.

Whether or not the client of an object is aware of the persistent state is a choice the object has. CORBA already provides a persistent reference handling interface (i.e., object_to_string, string_to_object, release, etc.). We expect that this will be sufficient for most clients to manage persistence of their referenced objects. But, because certain kinds of flexibility require the client to manage reference objects' persistence, the Persistent Object Service defines object interfaces for doing so. If this flexibility is not required, then these interfaces need not be supported or used.

The size, structure, access patterns and other properties of the dynamic and persistent state of the object varies tremendously. For many objects, their primary semantics are the efficient storage and access of its state for particular purposes. It is critical that the Persistent Object Service be able to support greatly different styles of usage and implementation in order to be useful to as many objects as possible.

As usual for object services, the primary task of this persistence specification is to define the interfaces that are needed to use the Persistent Object Service, and the conventions for how objects can work together using it.

The architecture of the Persistent Object Service defines multiple components and interfaces. In a particular situation, different parts of the service may be used. In no case does this specification assume the use of a particular implementation of a component, and it is expected that different implementations of the components will in fact work together.

Section 5.2 describes the overall goals and properties of the Persistent Object Service. Section 5.3 defines the components which compose it. Section 5.4 presents the CosPersistencePID module which defines the Persistence Identifier (PID). Section 5.5 presents the CosPersistencePO module with interfaces borne by Persistent Objects, and Section 5.6 presents the interface to the Persistent Object Manager (POM). Section 5.7 presents an overview of the Persistent Data Service (PDS) which interfaces both to the Protocol which communicates between PO and PDS, and to the Datastore which actually stores the data; following this, Section 5.8 defines the CosPersistencePDS Module which defines base functionality inherited by every protocol. Three protocols are presented in this specification although more are possible; the Direct Access

Protocol (PDS_DA) is described in Section 5.9 and its IDL module is presented in Section 5.10. The ODMG-93 Protocol is described in Section 5.11. The Dynamic Data Object (DDO) Protocol is described in Section 5.12, and its IDL module is presented in Section 5.13. Other possible protocols are discussed briefly in Section 5.14. One possible datastore, implementable using a number of database and file mechanisms, is described in Section 5.15; other possible datastores are discussed in Section 5.16. Finally, Section 5.18 lists outside works referenced in this chapter.

5.2 Goals and Properties

The Persistent Object Service plays a key role in structuring the object system. The model of how many objects work is critically dependent on consistent and integrated use of persistence. Like other object services, the Persistent Object Service provides interfaces that can support different implementations in order to obtain different qualities of service. Those interfaces allow different components to work together.

The overall persistence architecture has multiple components. Each will be introduced in turn in this section, following presentation of some basic capabilities and properties provided by the overall architecture.

5.2.1 Basic Capabilities

The principle requirement to be supported is the need for an object to be able to make all or part of its state be persistent. Although the CORBA system defines object references as persistent (that is, they are usable until they are released regardless of the life time of their containing address space), it defined no particular way for the object to make its state persistent. The Persistent Object Service is intended ultimately to be the most common way to implement this. Therefore, there must be a way for the object to decide what state needs to be made persistent, and ways to store and retrieve that state.

It is often necessary to expose the persistent state from an object, so that the client can control the object's persistence to achieve certain types of flexibility. The Persistent Object Service defines a convention for doing this. Clients of objects sometimes need ways to refer to the persistent state, and request various operations on it. It is often not necessary to expose the persistent state from an object, so that the object implementation itself determines its persistence. In these cases, no persistence-specific object interfaces need be supported.

5.2.2 Object-oriented Storage

In existing non-object-oriented systems, persistence is accomplished by a number of data storage mechanisms. Generally, such mechanisms do not provide the key properties that object systems provide—uniform interfaces, self-description, and abstraction. The Persistent Object Service brings these properties to storage by applying object technology and principles.

Interfaces to Data

To manage object persistence, the POS defines an architecture with interfaces defined using the CORBA IDL type system. Whether detailing the particular data to be stored, describing the protocol for accessing the state, or defining the convention for making state visible for client control, the same “language” is used. This makes persistence a natural part of the software environment. These interfaces are designed to be used in a wide variety of situations, creating uniformity by encouraging most objects to support them, while allowing optimization and evolution.

By accessing data through an interface, many problems of data manipulation and exchange can be avoided. For example, programs always see data in the representation that is appropriate for the machine, programming language, etc., of the application. Data can be translated as needed to facilitate use in different object types and implementations and for different storage formats or underlying persistent storage mechanisms (e.g. stream files, record files, or various databases) when it is accessed through the interface.

Self-description

A powerful characteristic of object-oriented systems is that the elements are self-describing. It is possible to determine from an object what kind of object it is and what interfaces it supports. In the persistence architecture this means, for example, that a client can determine whether or not an object wishes to make its persistent state visible by checking to see if the object supports the interface for doing so.

It also means that the data can be manipulated to some degree independently of the objects whose state they represent. This can allow generic facilities such as backup, migration, storage accounting, etc., to be done independent of the objects whose state is being stored.

Abstraction

In order to support a wide and evolving set of uses, a service must be able to improve and replace its implementations without affecting the clients of that service. The desire for reuse of objects requires that those objects not depend too strictly on other objects and services, but rather be willing to work with any other components that support the required interface.

A variety of value-added products are also possible assuming that the objects depend only on the defined interfaces. By interposing unexpected implementations, for example, it may be possible to support features such as replication or versioning in a transparent way.

5.2.3 Open Architecture

A major feature of the Persistent Object Service (and the OMG architecture) is its openness. In this case, that means that there can be a variety of different clients and implementations of the Persistent Object Service, and they can work together. This is particularly important for storage, where the mechanisms that are useful for documents may not be appropriate for employee databases, or the mechanisms appropriate for mobile computers may not be appropriate for mainframes.

Implementations can be lightweight, consisting of mostly library code, or powerful, leveraging decades of experience with database systems. Of course, the architecture specifies several interfaces, but also shows how new interfaces can be introduced when needed while still exploiting the rest of the architecture.

As with other object services, the Persistent Object Service is intended to be part of a collection of services. As a result, it does not attempt to solve all problems that might relate to storage. Rather, it assumes other services will provide the solutions. For example, the Persistent Object Service does not do naming, but assumes that the Name Service will perform that function; it does not do transactions, but assumes that they will be added as appropriate; it does not handle issues of general compound objects, but assumes that there will be a scheme that spans persistence, lifecycle, printing and other services.

A key idea in object systems that is critical for persistence is the ability for new and existing storage services to be able to integrate into the architecture. The requirement for such components to "plug and play" together is paramount, since one cannot expect all data to be maintained in a particular kind of file or database system. Thus, the architecture has features to allow existing databases or other storage mechanisms to be used for persistence, and for new storage mechanisms to be developed that can support both Persistent Object Service clients and other kinds of clients.

The POS architecture is open with respect to PersistentDataService, Datastore, Protocol, and PID interfaces. Although we define some minimum requirements for these in some cases, many alternatives are allowed, including ones that have not yet been defined.

5.2.4 Views of Service

There are multiple views of the service, and each participant may need to consider only a part of the architecture.

Client

It is common for clients of objects to need to control or to assist in managing persistence. In particular, the timing of when the persistent state is preserved or restored, and the identification of which persistent state is to be used for an object, are two aspects often of interest to clients. The ability of a client to see the object and its data separately allows different object implementations to be used with the same data and allows different files or databases and formats to be used with the same object implementation.

However, the client need only deal with such complexity when this type of functionality is necessary. The client of the object can be completely ignorant of the persistence mechanism, if the object chooses to hide it.

The Persistent Object Service provides an interface for objects to use when they want to expose their persistence to their clients. The interface does not completely abandon encapsulation, but gives the client visibility to those functions it needs. In fact, the client is generally unaware of how or if the object uses other parts of the Persistent Object Service.

Object Implementation

The object has the most involvement with the persistence, and the most options in deciding how to use it. Defining and manipulating the persistent state of the object is often the most crucial part of its implementation. The first decision the object makes is what interface to its data it needs. The Persistent Object Service captures that choice in the selection of the Protocol used by the object. Some Protocols provide simple interfaces and limited functionality, others may provide more control and more powerful operations.

The object also has the choice of delegating the management of its persistent data to other services, or maintaining fine-grained control over it. The Persistent Object Service defines a Persistent Object Manager that handles much of the complexity of establishing connections between objects and storage, allowing new components to be introduced without affecting the objects or their clients.

The object may also provide the ability for its clients to manipulate its persistent state in various ways. This is important for creating a uniform view of persistence in the system.

Persistent Data Service

The Persistent Data Service (PDS) actually implements the mechanism for making data persistent and manipulating it. A particular PDS supports a Protocol defining the way data is moved in and out of the object, and an interface to an underlying Datastore.

The PDS has the responsibility of translating from the object world above it to the storage world below it. It plays critical roles in identifying the storage as well as providing convenient and efficient access to it.

We define multiple kinds of PDSs, each tuned to a particular protocol and data storage mechanism, since the range of requirements for performance, cost, and qualitative features is so large. Multiple PDSs must work together to create the impression of a uniform persistence mechanism. The Persistent Object Manager provides the framework for PDSs to cooperate this way.

Datastore

The lowest-level interface we define is a Datastore. Although Datastore interfaces are the least visible part of the persistence architecture, it may be the most valuable, since there are so many different Datastores offering a wide spectrum of tradeoffs between availability, data integrity, resource consumption, performance and cost, and it is expected that more will be created. By having an interface that is hidden from objects and their clients, a Datastore can provide service to any and all objects that indirectly use the Datastore interface.

The Datastore plays a key role in interoperating with other storage services. It is the manifestation in the object world of the various means of storing data that are not objects. Generally, standards for Datastore interfaces have already been defined for different kinds of data repositories - relational, object-oriented, and file systems.

5.3 Service Structure

This section presents an overview of each of the major components and how they interrelate. Subsequent sections present the OMG IDL as divided into modules which correspond closely (but not exactly) to these components, as noted below.

The major components of the Persistent Object Service are illustrated in Figure 5-1 on page 1. They are:

- Persistent Identifier (PID) - This describes the location of an object's persistent data in some Datastore and generates a string identifier for that data.
- Persistent Object (PO) - This is an object whose persistence is controlled externally by its clients.
- Persistent Object Manager (POM) - This component provides a uniform interface for the implementation of an object's persistence operations. An object has a single POM to which it routes its high-level persistence operations to achieve plug and play.
- Persistent Data Service (PDS) - This component provides a uniform interface for any combination of Datastore and Protocol, and coordinates the basic persistence operations for a single object.
- Protocol - This component provides one of several ways to get data in and out of an object.
- Datastore - This component provides one of several ways to store an object's data independently of the address space containing the object.

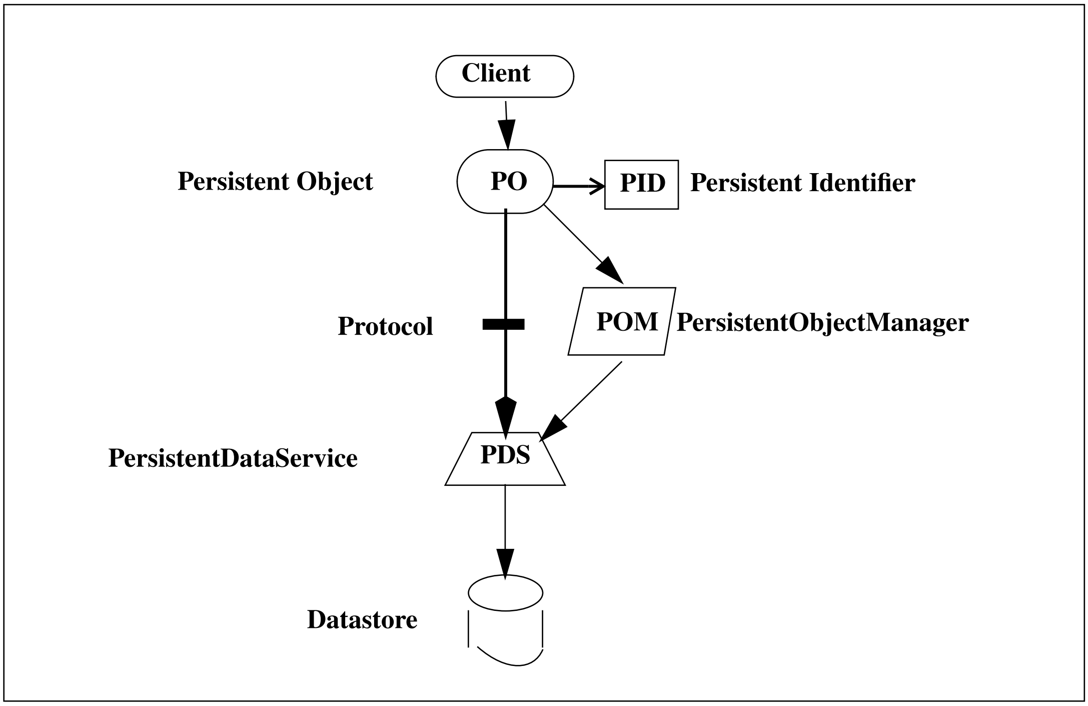


Figure 5-2 Major Components of the POS and their Interactions

The term "persistent object" is used to refer both to objects whose persistence is controlled internally or externally. Either kind of persistent object can be supported by the Persistent Object Service's POM, PDS, Protocol and Datastore interfaces. The PO interface supports externally controlled persistence.

5.4 The CosPersistencePID Module

The CosPersistencePID module contains the basic interface for retrieving a PID:

- The PID Interface

This section describes this interface, plus an example factory interface, and their operations in detail.

The CosPersistencePID Module is shown in Figure 5-3: .

```
module CosPersistencePID {

    interface PID {
        attribute string datastore_type;
        string get_PIDString();
    };

};
```

Figure 5-3 The CosPersistencePID Module

The PID identifies one or more locations within a Datastore that represent the persistent data of an object and generates a string identifier for that data. An object must have a PID in order to store its data persistently. The client can create a PID, initialize its attributes, and connect it to the object. A persistent object's implementation uses the POM interface by passing the object and the PID as parameters.

The PID should not be confused with the CORBA object reference (OID). They are similar in that both have an operation that produces a string form that can be stored or communicated in whatever ways strings may be manipulated and later used to get the original PID or OID. They differ in that the PID identifies data while the OID identifies a CORBA object.

For example, assume mySpreadSheet object is referenced by both myDoc and yourDoc objects. If mySpreadSheet's OID is stored persistently with myDoc and yourDoc and then all three are brought into memory, then both documents will always see the same spreadsheet object. If mySpreadSheet's PID is stored persistently with myDoc and yourDoc and then all three object are brought into memory, each document will see a different spreadsheet object whose states will be the same initially but will diverge over time.

5.4.1 PID Interface

The OMG IDL definition for the PID is as follows

```
interface PID {
    attribute string datastore_type;
    string get_PIDString();
 };
```

The PID contains at least one attribute:

attribute string datastore_type;

This identifies the interface of a Datastore. Example datastore_types might be “DB2”, “PosixFS” and “ObjectStore”. The PDS hides the Datastore’s interface from the client, the ppersistent object and the POM, but PDS implementations are dependent on the Datastore’s interface.

Other attributes can be added via subtyping the PID base type to reflect more specialized PIDs. Unless the datastore_type contains only a single object’s persistent data, there is a need for more specific location information in the PID. The following example PID subtypes illustrate this:

```
#include "CosPersistencePID.idl"

interface PID_DB : CosPersistencePID::PID {
	attribute string database_name; // name of a database
};

interface PID_SQLDB : PID_DB {
	attribute string sql_statement; // SQL statement
};

interface PID_OODB : PID_DB {
	attribute string segment_name;// segment within database
	attribute unsigned long oid; //object id within a segment
};
```

The PID provides a single operation:

string get_PIDString();

This operation returns a string version of the PID called the PIDString. A client should only obtain the PIDString using the get_PIDString operation. This allows the PID implementation to decide the form of the PIDString.

Some implementations may simply concatenate the PID attributes. Others may return a more compact form specialized for specific Datastores or even databases within a Datastore. Still others may return a universally unique identifier (UUID) that facilitates movement of its persistent data either within a single Datastore or between Datastores. A UUID-based PID might be implemented by overriding the get and set attribute operations and the get_PIDString operation to bind and lookup the mapping between UUID and location information in a special context in the Name Service. Using such a UUID-based PID, when an object is moved, the new location would be changed by setting the attributes to indicate the new location, and the PID would make the modification in the Name Service. The PIDString would contain the UUID that does not change when an object’s data is moved, so that references remain intact.

Some applications need to be able to restore an object given a PID but without knowing which type or implementation to use. The PID can be subtyped to accommodate this by adding the type or implementation as a PID attribute.

5.4.2 Example PIDFactory Interface

The OMG IDL definition for an example PIDFactory is as follows (others are also possible):

```
interface PIDFactory {
        CosPersistencePID::PID create_PID_from_key(in string key);
        CosPersistencePID::PID create_PID_from_string(
                in string pid_string);
        CosPersistencePID::PID create_PID_from_string_and_key(
                in string pid_string, in string key);
};
```

This example PIDFactory provides three ways of creating a PID:

CosPersistencePID::PID create_PID_from_key(in string key);
This creates an instance of a PID given a key that identifies a particular PID implementation.

CosPersistencePID::PID create_PID_from_string(in string pid_string);
This creates an instance of a PID given a PIDString. The PIDString must include some way to identify a particular PID implementation (the PID's key) in some way that allows this operation to extract the PID's key from the PIDString. This key identifies the PID implementation for the newly created PID.

CosPersistencePID::PID create_PID_from_string_and_key(in string pid_string, in string key);
This creates an instance of a PID whose implementation is identified by the key in the input parameter instead of the key in the PIDString, and whose value is determined by the PIDString. This is useful for when persistent data is moved between Datastores that require different PID interfaces.

5.5 The CosPersistencePO Module

The CosPersistencePO Module collects the interfaces which are borne by a persistent object to allow its clients and the POM to control the PO's relationship with its persistent data. This module includes two interfaces:

- The PO Interface
- The SD Interface

plus an example factory interface.

The PO interface is borne by the PO and used by the client. The SD interface is borne by the PO and used by the POM.

This section describes these interfaces and their operations in detail.

The CosPersistencePO Module is shown in Figure 5-4::

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePO {

	interface PO {
		attribute CosPersistencePID::PID p;
		CosPersistencePDS::PDS connect (
			in CosPersistencePID::PID p);
		void disconnect (in CosPersistencePID::PID p);
		void store (in CosPersistencePID::PID p);
		void restore (in CosPersistencePID::PID p);
		void delete (in CosPersistencePID::PID p);
	};

	interface SD {
		void pre_store();
		void post_restore();
	};
};
```

Figure 5-4 TheCosPersistencePO Module

5.5.1 The PO Interface

The PO interface provides two mechanisms for allowing a client to externally control the PO's relationship with its persistent data:

- Connection: This mechanism establishes a close relationship between the PO and its Datastore where the two data representations can be viewed as one for the duration of the connection. When the connection is ended, the data is the same in the PO and the Datastore, and the relationship between them no longer exists. An object can have only one connection at a time.
- Store/restore: These operations allow the client to move data between the PO and its Datastore in each direction separately, with each movement in each direction explicitly initiated by the client.

The PO interface operations allow client control of a single PO's persistent data. When one of these operations is performed on a PO, what data is included in these operations is up to that PO's implementation. For example, only part of the PO's private data may be included. Other POs may be included based on any criteria. If other POs are included, the target PO's implementation becomes their client and is responsible for controlling their persistence.

A PO client is responsible for the following:

- Creating a PID for the PO and initializing the PID. For storage, whatever location information is not specified will be determined by the Datastore. For a retrieval or delete operation, the location information must be complete.

- Controlling the relationship between the data in the PO and the Datastore. This is done by asking the PO to connect(), disconnect(), store(), restore() or delete() itself.

The OMG IDL definition for a PO is as follows:

```
interface PO {
        attribute CosPersistencePID::PID p;
        CosPersistencePDS::PDS connect (
                in CosPersistencePID::PID p);
        void disconnect (in CosPersistencePID::PID p);
        void store (in CosPersistencePID::PID p);
        void restore (in CosPersistencePID::PID p);
        void delete (in CosPersistencePID::PID p);
};
```

The PO interface has the following operations:

CosPersistencePDS::PDS connect (in CosPersistencePID::PID p);
: This begins a connection between the data in the PO and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that handles persistence for use by those Protocols that require the PO to call the PDS.

void disconnect (in CosPersistencePID::PID p);
: This ends a connection between the data in the PO and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.

void store (in CosPersistencePID::PID p);
: This copies the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.

void restore (in CosPersistencePID::PID p);
: This copies the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.

void delete (in CosPersistencePID::PID p);
: This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

To adhere to the plug and play philosophy, objects pass these requests through to the POM, so that the interface for PO parallels that of the POM. This delegation to the POM allows objects to change PDSs (combination of Datastore and Protocol) without changing their implementation.

5.5.2 The POFactory Interface

The OMG IDL definition for an example POFactory is as follows (others are also possible):

```
#include "CosPersistencePO.idl"
// CosPersistencePO.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl

interface POFactory {
	CosPersistencePO::PO create_PO (
		in CosPersistencePID::PID p,
		in string pom_id);
};
```

The example POFactory provides the following operation:

CosPersistencePO::PO create_PO(in CosPersistencePID::PID p, in string pom_id);
This creates an instance of a PO that knows which POM to use and with its pid attribute already assigned.

5.5.3 The SD Interface

Some objects may be implemented knowing they are going to be persistent. Many such objects have both transient and persistent data. The Synchronized Data (SD) Interface is provided to allow such objects to synchronize their transient and persistent data. Operations on the SD are invoked only by the POM. Persistent objects whose persistence is controlled either internally or externally (PO) can support the SD interface.

The OMG IDL definition for SD is as follows:

```
interface SD {
	void pre_store();
	void post_restore();
};
```

The interface for SD provides two operations:

void pre_store();
This ensures that the persistent data are synchronized with the transient data.

void post_restore();
This ensures that the transient data are synchronized with the persistent data.

A word processing document provides a good example of how these operations might be implemented. Suppose the document type is implemented with the following data:

- text buffer (persistent)

- attributes (persistent)
- text cache (transient)
- cursor location (transient)

The document could be implemented such that all work is done in the text cache. Then at store time, the text buffer needs to be updated, since it contains the actual data that will be stored. As such, the pre_store operation should be implemented such that any updates in the text cache are propagated to the text buffer. The post_restore operation should be implemented such that the text cache is inititialized with a state consistent with the text buffer.

5.6 The CosPersistencePOM Module

The CosPersistencePOM module contains the interface which is borne by the POM and used by the PO. It contains a single interface:

- The POM Interface

This section describes this interface and its operations in detail.

The CosPersistencePOM Module is shown in Figure 5-5:

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePOM {

	interface Object;
	interface POM {
		CosPersistencePDS::PDS connect (
			in Object obj,
			in CosPersistencePID::PID p);
		void disconnect (
			in Object obj,
			in CosPersistencePID::PID p);
		void store (
			in Object obj,
			in CosPersistencePID::PID p);
		void restore (
			in Object obj,
			in CosPersistencePID::PID p);
		void delete (
			in Object obj,
			in CosPersistencePID::PID p);
	};
};
```

Figure 5-5 The CosPersistencePOM Module

Clients of a PO will see the operations of the POM interface indirectly through the PO interface. The implementation of a persistent object with either externally or internally controlled persistence can use the POM interface. The POM provides a uniform interface across all PDSs, so different PDSs (combination of Datastore and Protocol) can be used without changing the object's implementation.

The OMG IDL definition of the POM is as follows:

```
interface POM {
        CosPersistencePDS::PDS connect (
                in Object obj,
                in CosPersistencePID::PID p);
        void disconnect (
                in Object obj,
                in CosPersistencePID::PID p);
        void store (
                in Object obj,
                in CosPersistencePID::PID p);
        void restore (
                in Object obj,
                in CosPersistencePID::PID p);
        void delete (
                in Object obj,
                in CosPersistencePID::PID p);
};
```

The POM interface has the following operations:

CosPersistencePDS::PDS connect (in Object obj, in CosPersistencePID::PID p);
: This begins a connection between data in the object and the Datastore location indicated by the PID. The persistent state may be updated as operations are performed on the object. This operation returns the PDS that is assigned the object's PID for use by those Protocols that require the PO to call the PDS.

void disconnect (in Object obj, in CosPersistencePID::PID p);
: This ends a connection between the data in the object and the Datastore location indicated by the PID. It is undefined whether or not the object is usable if not connected to persistent state. The PID can be nil.

void store (in Object obj, in CosPersistencePID::PID p);
: This gets the persistent data out of the object in memory and puts it in the Datastore location indicated by the PID. The PID can be nil.

void restore (in Object obj, in CosPersistencePID::PID p);
: This gets the object's persistent data from the Datastore location indicated by the PID and inserts it into the object in memory. The PID can be nil.

void delete (in Object obj, in CosPersistencePID::PID p);
: This deletes the object's persistent data from the Datastore location indicated by the PID. The PID can be nil.

The major function of the POM is to route requests to a PDS that can support the combination of Protocol and Datastore needed by the persistent object. To do this, the POM must know which PDSs are available and which Protocol and Datastore combinations they support. There are several possible ways that this information can be made available to a POM:

- How a Protocol is associated with an object. One possibility is for the client to set the Protocol for that object. Another possibility is for the Protocol to be associated with the object's type or implementation.
- How a POM finds out the set of available PDSs and which Protocol (or object type) and Datastores they support. One possibility is for the POM to find the information in a configuration file or a registry. Another possibility is to provide an interface to the POM for registering the information. The best or most natural technique may depend on the environment.

Because there are multiple ways to accomplish the above and more experience is needed to better understand whether there is a best way and what that might be, a POM interface for registering this information in the POM is not specified at this time.

When the POM is asked to store an object, the following steps logically occur:

1. From the PID, the POM gets the datastore_type attribute.
2. Regardless of how the Protocol is associated with the object, the POM uses the combination of Protocol and datastore_type to determine the PDS.
3. The POM passes the store request through to the PDS.
4. The PDS gets data from the object using a Protocol and stores the data in the Datastore.

The routing function of the POM serves to shield the client from having to know the details of how actual data storage/retrieval takes place. A client can change the repository of an object by changing the PID. The change will result in routing the next store/restore request to whatever the appropriate PDS is for the new Datastore.

Figure 5-6 illustrates an example of the routing logic for the storage of myDoc in a DB2 database. This figure and the following example steps assume that, for this POM, the Protocol is associated with object type:

1. The POM is asked to perform a store on myDoc with pid1.
2. The POM finds the datastore_type associated with pid1 (e.g., DB2).
3. The POM finds the object type of myDoc (e.g., document).
4. The POM determines that myDoc will use a particular PDS (e.g., pds1).
5. The POM routes the store/restore to pds1.
6. The PDS gets the persistent data using protocol1 and stores the data in the DB2 Datastore at pid1.

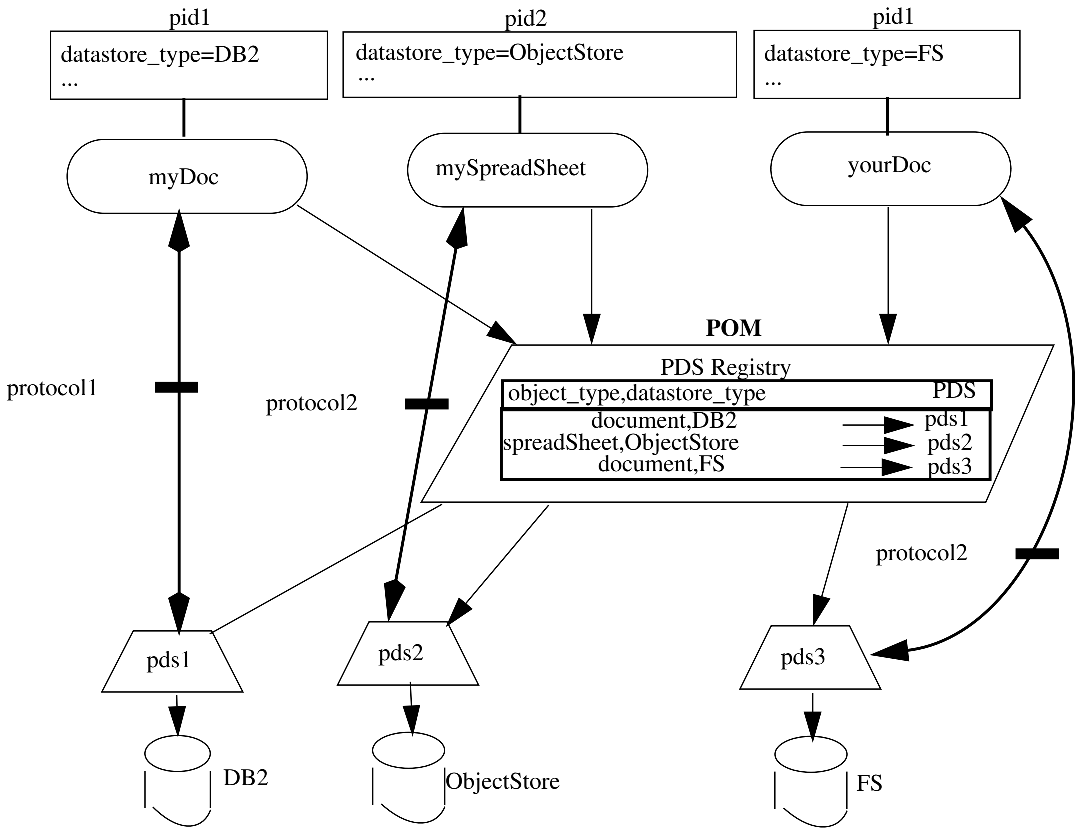


Figure 5-6 Example to illustrate POMFunctions

5.7 Persistent Data Service (PDS) Overview

The PDS implementation is responsible for the following:

- Interacting with the object to get data in and out of the object using a *protocol*. Protocols are introduced in this section; three example protocols and a discussion of additional protocols are presented in Section 5.9 through Section 5.14.
- Interacting with the Datastore to get data in and out of the object. Datastores are introduced in this section, and an example datastore plus a discussion of implementing additional datastores are presented in Section 5.15 and Section 5.16.

A PDS performs the work for moving data into and out of an object and moving data into and out of a Datastore. There can be a wide variety of implementations of PDSs which provide different performance, robustness, storage efficiency, storage format, or other characteristics, and which are tuned to the size, structure, granularity, or other properties of the object's state.

Because the range of storage requirements is so large, there may be different ways in which the object can best access its persistent data, and there may be different ways in which the PDS can store that data. The way in which the object interacts with the PDS is called the Protocol. A Protocol may consist of calls from the object to the PDS, calls from the PDS to the object, implicit operations implemented with hidden interfaces, or some combination. The interaction might be explicit, for example, asking the object to stream out its data, or implicit, for example, the object might be mapped into persistent virtual memory. The Protocol is initiated when an object's persistent state is stored, restored, or connected; this may be initiated by a POM or by the object itself. What happens after that depends on the particular Protocol. An object that uses a particular Protocol can work with any PDS that supports that Protocol. There is no "standard" protocol. This specification defines three Protocols: the Direct Attribute (DA) Protocol, the ODMG Protocol, and the Dynamic Data Object (DDO) Protocol. A PDS might also use a programming language-specific or runtime environment-specific or other Protocol.

A PDS may use either a standard or a proprietary interface to its Datastore. A Datastore might be a file, virtual memory, some kind of database, or anything that can store information. This specification defines one Datastore interface that can be implemented by a variety of databases (Section 5.15).

The PDS component interface is specified here as one module containing only the base PDS interface, plus one additional module per protocol. Each protocol-specific module inherits from the base module, augmenting the base functionality as needed.

5.8 The CosPersistencePDS Module

The CosPersistencePDS Module contains the base interface upon which protocol-specific interfaces are built. It contains a single interface: the PDS Interface.

This section describes this interface and its operations in detail.

The CosPersistencePDS module is shown in Figure 5-7. Some Protocols may require specialization of the PDS interface. However, no matter what Protocol or Datastore is used, a PDS always supports at least the following interface:

```
#include "CosPersistencePID.idl"

module CosPersistencePDS {

    interface Object;
    interface PDS {
        PDS  connect (in Object obj,
            in CosPersistencePID::PID p);
        void disconnect (in Object obj,
            in CosPersistencePID::PID p);
        void store (in Object obj,
            in CosPersistencePID::PID p);
        void restore (in Object obj,
            in CosPersistencePID::PID p);
        void delete (in Object obj,
            in CosPersistencePID::PID p);
    };
};
```

Figure 5-7 The CosPersistencePDS Module

The exact semantics of the connect, disconnect, store, and restore operations depend on the Protocol, since there may be other steps involved in the Protocol. In all four operations, the persistent state is determined by the PID of the object.

PDS connect (in Object obj, in CosPersistencePID::PID p);
    This connects the object to its persistent state, after disconnecting any previous persistent state. The persistent state may be updated as operations are performed on the object.

void disconnect (in Object obj, in CosPersistencePID::PID p);
    This disconnects the object from the persistent state. It is undefined whether or not the object is usable if not connected to persistent state.

void store (in Object obj, in CosPersistencePID::PID p);
    This saves the object's persistent state.

void restore (in Object obj, in CosPersistencePID::PID p);
    This loads the object's persistent state. The persistent state will not be modified unless a store or other mutating operation is performed on the persistent state.

void delete (in Object obj, in CosPersistencePID::PID p);
    This disconnects the object from its persistent state and deletes the object's persistent data from the Datastore location indicated by the PID.

5.9 The Direct Access (PDS_DA) Protocol

The first protocol to be described here is the PDS_DA or Direct Access Protocol. The Direct Access Protocol supports direct access to persistent data through typed attributes organized in data objects that are defined in a Data Definition Language (DDL). An object using this Protocol would represent its persistent data as one or more interconnected data objects. For uniformity, the persistent data of an object is described as a single data object; however, that data object might be the root of a graph of data objects interconnected by stored data object references. If an object uses multiple data objects, the object traverses the graph by following stored data object references.

An object must define the types of the data objects it uses. Those types are specified in DDL, which is a subset of the OMG Interface Definition Language (OMG IDL) in which objects consist solely of attributes. The state of the data object is accessed using the attribute access operations defined in CORBA in conjunction with the appropriate programming language mapping.

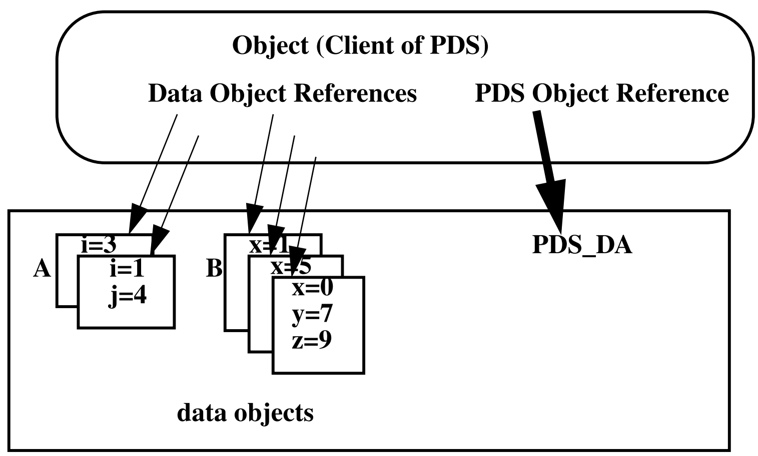


Figure 5-8 Direct Access Protocol Interfaces

The PDS_DA Protocol has two parts, as shown in Figure 5-8. When connected to a PDS, the object (which is effectively the client of the PDS) has an object representing the PDS which supports the PDS_DA interface. The object performs operations defined in the PDS_DA interface to get references to the data objects in the PDS. The persistent data is manipulated by performing operations using the data object references to get and set attributes on the collection of data objects in the PDS.

5.10 The CosPersistencePDS_DA Module

The CosPersistencePDS_DA Module is a collection of interfaces which together define the protocol. This module contains the following interfaces:

- The PID_DA Interface
- The DAObject Interface

- The DAObjectFactory Interface
- The DAObjectFactoryFinder Interface
- The PDS_DA Interface
- The DynamicAttributeAccess Interface
- The PDSClustered_DA Interface

This section describes these interfaces and their operations in detail.

The CosPersistencePDS_DA Module is shown in Figure 5-9: :

```
#include "CosPersistencePDS.idl"
// CosPersistencePDS.idl #includes CosPersistencePID.idl

module CosPersistencePDS_DA {

	typedef string DAObjectID;

	interface PID_DA : CosPersistencePID::PID {
		attribute DAObjectID oid;
	};

	interface DAObject {
		boolean dado_same(in DAObject d);
		DAObjectID dado_oid();
		PID_DA dado_pid();
		void dado_remove();
		void dado_free();
	};

	interface DAObjectFactory {
		DAObject create();
	};

	interface DAObjectFactoryFinder {
		DAObjectFactory find_factory(in string key);
	};

	interface PDS_DA : CosPersistencePDS::PDS {
		DAObject get_data();
		void set_data(in DAObject new_data);
		DAObject lookup(in DAObjectID id);
		PID_DA get_pid();
		PID_DA get_object_pid(in DAObject dao);
		DAObjectFactoryFinder data_factories();
	};
```

Figure 5-9 The CosPersistencePDS_DA Module

```
        typedef sequence<string> AttributeNames;
        interface DynamicAttributeAccess {
                AttributeNames attribute_names();
                any attribute_get(in string name);
                void attribute_set(in string name, in any value);
        };

        typedef string ClusterID;
        typedef sequence<ClusterID> ClusterIDs;
        interface PDS_ClusteredDA : PDS_DA{
                ClusterID cluster_id();
                string cluster_kind();
                ClusterIDs clusters_of();
                PDS_ClusteredDA create_cluster(in string kind);
                PDS_ClusteredDA open_cluster(in ClusterID cluster);
                PDS_ClusteredDA copy_cluster(
                        in PDS_DA source);
        };
};
```

Figure 5-9 The CosPersistencePDS_DA Module

5.10.1 The PID_DA Interface

The Persistent Identifiers (PIDs) used by the PDS_DA contain an object identifier that is local to the particular PDS. This value may be accessed with the following extension to the CosPersistencePID interface:

```
interface PID_DA : CosPersistencePID::PID {
        attribute DAObjectID oid;
};
```

The DAObjectID has the following attribute:

attribute DAObjectID oid();

This returns the data object identifier used by this PDS for the data object specified by the PID.The DAObjectID type is defined as an unbounded sequence of bytes that may be vendor-dependent.

5.10.2 The Generic DAObject Interface

The DAObject interface defined below provides operations that many data object clients need. A Datastore implementation may provide support for these operations automatically for its data objects. A data object is not required to support this interface. A client can obtain access to these operations by narrowing a data object reference to the DAObject interface:

```
interface DAObject {
        boolean dado_same(in DAObject d);
        DAObjectID dado_oid();
        PID_DA dado_pid();
        void dado_remove();
        void dado_free();
};
```

The DAObject has the following operations:

boolean dado_same(in DAObject d);
This returns true if the target data object and the parameter data object are the same data object. This operation can be used to test data object references for identity.

DataObjectID dado_oid();
This returns the object identifier for the data object. The scope of data object identifiers is implementation-specific, but is not guaranteed to be global.

PID_DA dado_pid();
This returns a PID_DA for the data object.

void dado_remove();
This deletes the object from the persistent store and deletes the in-memory data object.

void dado_free();
This informs the PDS that the data object is not required for the time being, and the PDS may move it back to persistent store. The data object must be preserved and must be brought back the next time it is referenced. This operation is only a hint and is provided to improve performance and resource usage.

5.10.3 The DAObjectFactory Interface

The scheme for factories is consistent with that of the Life Cycle Service. The factory supports the following interface:

```
interface DAObjectFactory {
        DAObject create();
};
```

The DAObjectFactory has the following operation:

DAObjectFactory create();
creates a new data object in the PDS.

5.10.4 The DAObjectFactoryFinder Interface

This scheme for factories follows the Life Cycle Services specification. The factory finder supports the following interface:

```
interface DAObjectFactoryFinder {
	DAObjectFactory find_factory(in string key);
};
```

The DAObjectFactoryFinder has the following operation:

DAObjectFactoryFinder find_factory(in string key);
This finds a factory for data objects as specified by the key.

5.10.5 The PDS_DA Interface

The DA Protocol uses an extended PDS interface called PDS_DA:

```
interface PDS_DA : CosPersistencePDS::PDS {
	DAObject get_data();
	void set_data(in DAObject new_data);
	DAObject lookup(in DAObjectID id);
	PID_DA get_pid();
	PID_DA get_object_pid(in DAObject dao);
	DAObjectFactoryFinder data_factories();
};
```

The PDS_DA provides the following operations:

DAObject get_data();
This returns the single root data object of the PDS.

void set_data(in DAObject new_data);
This sets the single root data object

DAObject lookup(in DAObjectID id);
This finds a data object by object id.

PID_DA get_pid();
This constructs a PID that corresponds to the single root data object of this PDS.

PID_DA get_object_pid(in DAObject dao);
This constructs a PID that corresponds to the specified data object, which must be in this PDS.

DAObjectFactoryFinder data_factories();

This returns a factory finder. The factory finder will provide factories for the creation of new data objects within the PDS.

5.10.6 Defining and Using DA Data Objects

A PDS_DA implements data objects that have a set of attributes defined in a Data Definition Language (DDL). DDL is a subset of OMG IDL. In DDL, all interfaces consist only of attributes; that is, there are no operations. The programming interface for accessing the persistent state is the CORBA-defined attribute access operations as specified in the particular programming language mapping. A PDS_DA implements those accessor operations and transfers the persistent state between the Datastore and data objects as necessary.

DA data objects are used like normal CORBA objects. They are manipulated using object references, sometimes called “data object references”. Language mappings to data object interfaces are generated just like language mappings for other interfaces.

To define a DA data object (DADO), the developer decides what state must be made persistent. For example, suppose the object’s persistent data consists of two values, one integer and one floating point number. The developer would define a data object interface MyDataObject describing this data:

```
interface MyDataObject {
        attribute short my_short;
        attribute float my_float;
};
```

The DDL definition must be compiled, installed and linked with the object implementation as necessary for the particular PDS and CORBA environment. Mechanisms similar to those for creating stubs for IDL interfaces are used to provide the callable routines and create the runtime information necessary for the PDS implementation. The precise mechanisms are not defined in this specification.

Once the object has been connected to the PDS, the factory operations described above are used to create the data object and set it as the root object in the PDS. The object gets or sets values for the attributes using the CORBA accessor operations, for example:

```
// PDS_DA Examples
// C++ code
// Include IDL compiler output from CosPersistencePDS_DA.idl
#include "CosPersistencePDS_DA.xh"
// CosPersistencePDS_DA.idl #includes CosPersistencePDS.idl
// CosPersistencePDS.idl #includes CosPersistencePID.idl
// connect to PDS
CosPersistencePDS_DA::PDS_DA my_pds =
        pom->connect(my_object,my_PID);
// get factory finder
DAObjectFactoryFinder daoff = my_pds->data_factories();
// get factory for MyDataObject
DAObjectFactory my_factory =
        daoff->find_factory("MyDataObject");
// create an instance of MyDataObject
MyDataObjectRef my_obj = my_factory->create();
// set the object to be the root object
my_pds->set_data(my_obj);
// put persistent state in attributes
my_obj->my_short(42);
my_obj->my_float(3.14159);
// use persistent state
my_obj->my_short(my_obj->my_short()+12);
```

The DA Protocol allows developers to build simple object implementations that just read and write attribute values whenever they need to. There is no need for an object to cache persistent data in its transient store or to explicitly request it to be read or written.

Attributes can be defined using the full flexibility of the DDL type system. A particular PDS may restrict the attribute types it supports.

A data object may contain object references to other data objects and to ordinary CORBA objects. Here is an example that extends the previous example by adding a data object reference attribute and an ordinary CORBA object reference:

```
interface MyDataObject {
    attribute short my_short;
    attribute float my_float;
    attribute MyDataObject next_data;
    attribute SomeOtherObject my_object_ref;
};
```

This example allows an instance of MyDataObject to refer to another instance. A Datastore implementation might restrict the scope of stored data object references. For example, it might permit only references to data objects in the same Datastore.

DDL interfaces support inheritance with semantics identical to IDL. In the following example, a new type of data object is defined that has all the attributes of MyDataObject, plus an additional integer:

```
interface DerivedObject : MyDataObject {
        attribute short my_extra;
};
```

Like other CORBA objects, data objects support operations on object references. In particular, the get_interface operation, which returns an interface repository reference to the object's most derived interface, is useful for dynamically determining the type of a data object.

5.10.7 The DynamicAttributeAccess Interface

Because data objects are CORBA objects, the CORBA Dynamic Invocation Interface can be used to get and set data object attributes dynamically, using strings to identify attributes at run time. However, to simplify dynamic access to data object attributes, the DynamicAttributeAccess interface is defined. This interface defines operations that allow determination of the names of the attributes of a data object and getting and setting individual attribute values by name. A data object is not required to support this interface. It can be determined whether or not a data object supports these operations by narrowing a data object reference to the DynamicAttributeAccess interface.

```
typedef sequence<string> AttributeNames;
interface DynamicAttributeAccess {
        AttributeNames attribute_names();
        any attribute_get(in string name);
        void attribute_set(in string name, in any value);
};
```

AttributeNames attribute_names();
This returns a sequence containing the names of the object's attributes.

any attribute_get(in string name);
This returns the value of the specified attribute.

void attribute_set(in string name, in any value);
This sets the value of the named attribute to the value specified by the any parameter.

5.10.8 The PDS_ClusteredDA Interface

It is often useful to group data objects together within a PDS. Common reasons include locking, sharing, performance, etc. The PDS_ClusteredDA is an extension to the PDS_DA. A non-clustered PDS_DA is effectively a single cluster.

Each cluster is represented as a distinct instance of the PDS_ClusteredDA interface, although they will typically all be implemented by the same service using the same Datastore.

In addition to supporting the normal PDS_DA interface, a Clustered PDS_DA has the following interface:

```
typedef string ClusterID;
typedef sequence<ClusterID> ClusterIDs;
interface PDS_ClusteredDA : PDS_DA {
        ClusterID cluster_id();
        string cluster_kind();
        ClusterIDs clusters_of();
        PDS_ClusteredDA create_cluster(in string kind);
        PDS_ClusteredDA open_cluster(in ClusterID cluster);
        PDS_ClusteredDA copy_cluster(
                in PDS_DA source);
};
```

ClusterID cluster_id();
This returns the id of this cluster.

string cluster_kind();
This returns the kind of this cluster.

ClusterIDs clusters_of();
This returns a sequence of ClusterIDs listing all of the clusters in this Datastore.

PDS_ClusteredDA create_cluster(in string kind);
This creates a new cluster of the specified kind in this Datastore and returns a PDS_ClusteredDA instance to represent it.

PDS_ClusteredDA open_cluster(in ClusterID cluster);
This opens an existing cluster that has the specified ClusterID.

PDS_ClusteredDA copy_cluster(in PDS_DA source);
creates a new cluster, loading its state from the specified cluster, which may be implemented in a different Datastore.

5.11 The ODMG-93 Protocol

A group of Object-Oriented Database Management System (ODBMS) vendors has recently endorsed and published a common ODBMS specification called ODMG-93. That specification defines an extended version of IDL for defining ODBMS object types as well as programming language interfaces for object manipulation.

The ODMG-93 Protocol is similar to the DA Protocol, in that the object accesses attributes organized as data objects. The primary difference is that the ODMG-93 Protocol uses the Object Definition Language (ODL) defined in ODMG-93 instead of DDL, and it uses the programming language mapping defined for data objects specified in ODMG-93, rather than the CORBA IDL attribute operations.

If the ODMG-93 database object inherits the PDS_DA interface, then the database object can be used with the rest of this specification. Objects using the ODMG-93 Protocol would manipulate persistent data using the interfaces specified in ODMG-93.

Note that in addition to using the ODMG-93 interface as another protocol, it would be straightforward to implement the DA Protocol using an ODMG-93 ODBMS as a PDS. Since the DA Protocol is a subset of the functionality in ODMG-93, in most programming languages the language mapping for the DDL attributes would be a trivial layer on the ODMG-93 mapping. Using the ODMG-93 Protocol would fully exploit the capabilities of ODMG-93; using an ODMG-93 ODBMS to implement the DA Protocol captures those objects that use DA Protocol.

5.12 The Dynamic Data Object (DDO) Protocol

The DDO is a Datastore-neutral representation of an object's persistent data. Its purpose is to contain all of the data for a single object. Figure 5-1 illustrates an example of a DDO. A DDO has a single PID, object_type and set of data items whose cardinality is data_count. Each piece of data has a data_name, data_value and a set of properties whose cardinality is property_count. Each property has a property_name and a property value.

Although any data can be stored in a DDO, the following example illustrates how it might map onto a row in a table:

- a DDO = a row
- data_count = number of rows
- data_item = column
- data_name = column name
- data_value = column value
- property_count = number of column properties
- property_name = e.g., type or size
- property_value = e.g., character or 255

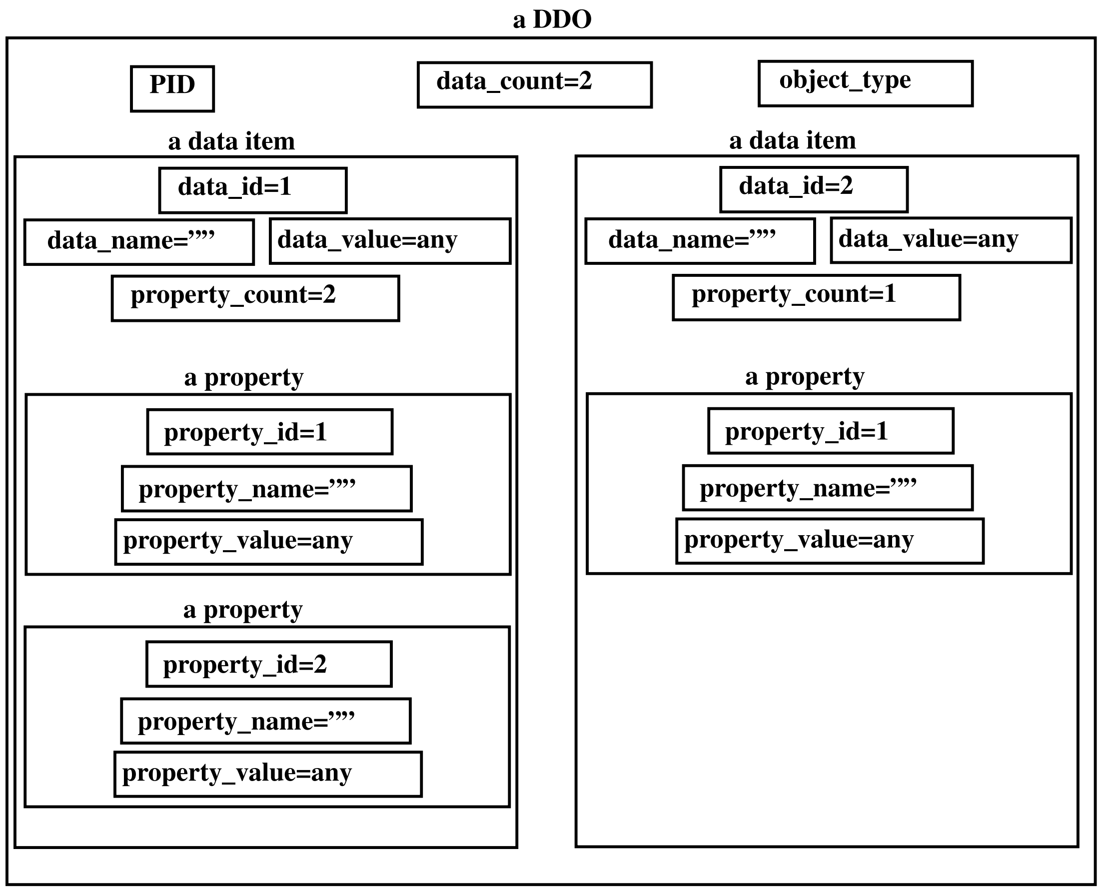


Figure 5-10 Structure of a DDO

A DDO provides a Protocol when the persistent object supports the DDO interface. In this case, the DDO interface is used to get data in and out of the persistent object. It may even provide the way that the persistent object stores its internal data, in which case a copy and reformat step is avoided.

To facilitate fast and simple storage and retrieval in specialized types of Datastore, DDOs can be used with particular conventions that are more suitable to different types of Datastore. If the DDO is used for both a Protocol and as a direct way to get data in and out of a Datastore, then copy and format costs are greatly reduced.

5.13 The CosPersistenceDDO Module

The CosPersistenceDDO module contains the OMG IDL to support the DDO protocol. The module contains oneinterface, the DDO interface.

This section describes the CosPersistenceDDO module in detail.

The CosPersistenceDDO Module is shown in Figure 5-11.

```
#include "CosPersistencePID.idl"

module CosPersistenceDDO {

	interface DDO {
		attribute string object_type;
		attribute CosPersistencePID::PID p;
		short add_data();
		short add_data_property (in short data_id);
		short get_data_count();
		short get_data_property_count (in short data_id);
		void get_data_property (in short data_id,
			in short property_id,
			out string property_name,
			out any property_value);
		void set_data_property (in short data_id,
			in short property_id,
			in string property_name,
			in any property_value);
		void get_data (in short data_id,
			out string data_name,
			out any data_value);
		void set_data (in short data_id,
			in string data_name,
			in any data_value);
	};
};
```

Figure 5-11 The CosPersistenceDDO Module

A DDO has two attributes:

attribute string object_type;
This identify the object_type that this DDO is associated with.

attribute CosPersistencePID::PID p;
This identify the PID of the DDO.

A DDO has the following operations for getting data in and out of the DDO:

short add_data();
This adds a new data item and returns a new data_id that can be used to access it.

short add_data_property (in short data_id);
This adds a new property within the data item identified by data_id and returns the new property_id that can be used to access it within the context of the data item.

short get_data_count();
This gets the number of data items in the DDO.

```
short get_data_property_count (in short data_id);
```
This gets the number of properties associated with the data item identified by data_id.

```
void get_data_property (in short data_id,
            in short property_id,
            out string property_name,
            out any property_value);
```
This gets the name and value of the property identified by property_id within the data item identified by data_id.

```
void set_data_property (in short data_id,
            in short property_id,
            in string property_name,
            in any property_value);
```
This sets the name and value of the property identified by property_id within the data item identified by data_id.

```
void get_data (in short data_id,
            out string data_name,
            out any data_value);
```
This gets the name and value of the data item identified by data_id.

```
void set_data (in short data_id,
            in string data_name,
            in any data_value);
```
This sets the name and value of the data item identified by data_id.

5.14 Other Protocols

This specification includes three protocols, but other protocols can be supported in this architecture. The proliferation of protocols would reduce the commonality of different objects, so it is desirable to use an existing protocol if that is possible. However, when a new protocol is required, it is still possible to use other parts of the Persistent Object Service with it. In general, the protocol should be independent of the Datastore interface, although some Datastore interfaces will be better suited to some protocols.

Some protocols are already defined and are not specified here. Such standard interfaces as POSIX files are already in wide use, and there is no need to respecify them. In this case, the PID would include the file name, and the protocol would consist of reads and writes.

Other protocols are intended to be value-added and non-standard. For example, a LISP-specific PDS might take advantage of knowledge of the LISP runtime environment to create the appearance of a single-level store of LISP objects. Although such a PDS would not be usable from other programming languages, it could provide significant value to LISP programmers. Of course, it is also possible for a particular value-added protocol to be implemented as a layer on a standard Protocol.

This specification allows such protocols to be integrated in the overall POS architecture without changing that architecture.

5.15 Datastores: CosPersistenceDS_CLI Module

The last major component in the architecture is a DataStore, which provides operations on a data repository underneath the Protocols just discussed. As with Protocols, a variety of DataStore interfaces may be defined. There is no “standard” DataStore interface. Only one kind of DataStore is defined here, for record-oriented databases, because other standard interfaces already exist at this level and many customers may choose to omit this level of the architecture altogether for performance in an object-oriented database by using the DA or ODMG Protocol directly on the DBMS.

Datastore_CLI provides a uniform interface for accessing many different Datastores either individually or simultaneously. The acronym CLI refers to the X/Open Data Management Call Level Interface on which the module is based. Datastore_CLI is especially suited for record database and file systems (e.g., relational, IMS, hierarchical databases, and VSAM file systems) that support user sessions, connections, transactions, and scanning through data items using cursors.

The specification of this framework, where appropriate, is consistent with the X/Open CLI, IDAPI, and ODBC standards. These are industry standards which specify procedure-oriented application programming interfaces for accessing data stored in any type of Datastore.

More detailed explanations and enumeration of the options in the Datastore_CLI operations can be found in the X/Open CLI Specification.

DDOs are used as the way data are passed into the Datastore_CLI interface. If DDO is also being used as the Protocol, the PDS can use this DDO directly as a parameter to calls to the Datastore_CLI. When a different Protocol is being used, the PDS must create a new DO and populate it with data prior to calling the Datastore_CLI.

The CosPersistenceDS_CLI module contains the interfaces derived from ODBC and IDAPI, providing cursors into relational and other databases. The module contains the following interfaces:

- The UserEnvironment Interface
- The Connection Interface
- The ConnectionFactory Interface
- The Cursor Interface
- The CursorFactory Interface
- The PID_CLI Interface
- The Datastore_CLI Interface

This section describes these interfaces and their operations in detail.

The CosPersistenceDS_CLI Module is shown in Figure 5-12:

```
#include "CosPersistenceDDO.idl"
// CosPersistenceDDO.idl #includes CosPersistencePID.idl

module CosPersistenceDS_CLI {
	interface UserEnvironment {
		void set_option (in long option,in any value);
		void get_option (in long option,out any value);
		void release();
	};

	interface Connection {
		void set_option (in long option,in any value);
		void get_option (in long option,out any value);
	};

	interface ConnectionFactory {
		Connection create_object (
			in UserEnvironment user_envir);
	};

	interface Cursor {
		void set_position (in long position,in any value);
		CosPersistenceDDO::DDO fetch_object();
	};

	interface CursorFactory {
		Cursor create_object (
			in Connection connection);
	};

	interface PID_CLI : CosPersistencePID::PID {
		attribute string datastore_id;
		attribute string id;
	};
```

```
        interface Datastore_CLI {
                void connect (in Connection connection,
                        in string datastore_id,
                        in string user_name,
                        in string authentication);
                void disconnect (in Connection connection);
                Connection get_connection (
                        in string datastore_id,
                        in string user_name);
                void add_object (in Connection connection,
                        in CosPersistenceDDO::DDO data_obj);
                void delete_object (
                        in Connection connection,
                        in CosPersistenceDDO::DDO data_obj);
                void update_object (
                        in Connection connection,
                        in CosPersistenceDDO::DDO data_obj);
                void retrieve_object(
                        in Connection connection,
                        in CosPersistenceDDO::DDO data_obj);
                Cursor select_object(
                        in Connection connection,
                        in string key);
                void transact (in UserEnvironment user_envir,
                        in short completion_type);
                void assign_PID (in PID_CLI p);
                void assign_PID_relative (
                        in PID_CLI source_pid,
                        in PID_CLI target_pid);
                boolean is_identical_PID (
                        in PID_CLI pid_1,
                        in PID_CLI pid_2);
                string get_object_type (in PID_CLI p);
                void register_mapping_schema (in string schema_file);
                Cursor execute (in Connection connection,
                        in string command);
        };

};
```

Figure 5-12 The CosPersistenceDS_CLI Module

5.15.1 The UserEnvironment Interface

The UserEnvironment OMG IDL is as follows:

```
interface UserEnvironment {
        void set_option (in long option,in any value);
        void get_option (in long option,out any value);
        void release();
};
```

The UserEnvironment has the following operations:

void set_option (in long option, in any value);
This sets the option to the desired value. The list of settable options is specified in the X/Open CLI Specification and the IDAPI Specification.

void get_option (in long option, out any value);
This gets the value of the option. The list of gettable options is the same as that for set_option().

void release();
This releases all resources associated with the UserEnvironment.

5.15.2 The Connection Interface

The Connection OMG IDL is as follows:

```
interface Connection {
        void set_option (in long option,in any value);
        void get_option (in long option,out any value);
};
```

The Connection interface contains the following operations:

void set_option (in long option,in any value);
This sets the option to the desired value. The list of settable options is specified in the IDAPI Specification.

void get_option (in long option, out any value);
This gets the value of the option. The list of gettable options is the same as that for set_option.

5.15.3 The ConnectionFactory Interface

The ConnectionFactory OMG IDL is as follows:

```
interface ConnectionFactory {
        Connection create_object (
                in UserEnvironment user_envir);
};
```

The ConnectionFactory has the following operation:

Connection create_object (
in UserEnvironment user_envir);
This creates an instance of Connection. A Connection is created within the context of a single UserEnvironment.

5.15.4 The Cursor Interface

The Cursor OMG IDL is as follows:

```
interface Cursor {
        void set_position (in long position,in any value);
        CosPersistenceDDO::DDO fetch_object();
};
```

A cursor is a movable pointer into a list of DDOs, through which a client can move about the list or fetch a DDO from the list. The Cursor has the following operations:

void set_position (in long position, in any value);
This sets the Cursor position to the desired value. The list of settable positions is specified in the IDAPI Specification.

CosPersistenceDDO::DDO fetch_object();
This fetches the next DDO from the list, based on the current position of the Cursor.

5.15.5 The CursorFactory Interface

The CursorFactory OMG IDL is as follows:

```
interface CursorFactory {
        Cursor create_object (
                in Connection connection);
};
```

The CursorFactory has the following operations:

Cursor create_object (in Connection connection);
This create an instance of Cursor. A Cursor is created within the context of a single Connection. See the X/Open CLI Specification and IDAPI Specification for more information.

5.15.6 The PID_CLI Interface

The PID_CLI IDL is as follows:

```
interface PID_CLI : CosPersistencePID::PID {
        attribute string datastore_id;
        attribute string id;
};
```

PID_CLI subtypes the PID base type (see Section 5.4.1), adding attributes required for the Datatstore_CLI interface. The PID_CLI interface has the following attributes:

attribute string datastore_id;

This identifies the specific datastore in use. Most datastore products support multiple datastores. For a relational database, this might be the name of a particular database containing multiple tables. For a Posix file system, this might be the pathname of a file.

attribute string id;

This identifies a particular data element within a datastore. For a relational database, this might be a table name and primary key indicating a particular row in a table. For a Posix file system, this might be a logical offset within the file indicating where the data starts.

5.15.7 The Datastore_CLI Interface

The Datastore_CLI OMG IDL is as follows:

```
interface Datastore_CLI {
        void connect (in Connection connection,
                in string datastore_id,
                in string user_name,
                in string authentication);
        void disconnect (in Connection connection);
        Connection get_connection (
                in string datastore_id,
                in string user_name);
        void add_object (in Connection connection,
                in CosPersistenceDDO::DDO data_obj);
        void delete_object (
                in Connection connection,
                in CosPersistenceDDO::DDO data_obj);
        void update_object (
                in Connection connection,
                in CosPersistenceDDO::DDO data_obj);
        void retrieve_object(
                in Connection connection,
                in CosPersistenceDDO::DDO data_obj);
        Cursor select_object(
                in Connection connection,
                in string key);
        void transact (in UserEnvironment user_envir,
                in short completion_type);
        void assign_PID (in PID_CLI p);
        void assign_PID_relative (
                in PID_CLI source_pid,
                in PID_CLI target_pid);
        boolean is_identical_PID (
                in PID_CLI pid_1,
                in PID_CLI pid_2);
        string get_object_type (in PID_CLI p);
        void register_mapping_schema (in string schema_file);
        Cursor execute (in Connection connection,
                in string command);
};
```

In general, a client goes through the following steps to store, restore or delete DDOs:

1. Create a UserEnvironment and set the appropriate options to their desired values.
2. Create a Connection and set the appropriate options to their desired values. Open a connection to the Datastore, via connect().
3. To store a DDO, call add_object() or update_object(). To restore a DDO, call retrieve_object(). To delete a DDO, call delete_object().
4. If necessary, call transact() to commit or abort a Datastore transaction.

5. Repeat steps 3 and 4 as necessary.

6. Close the connection to the Datastore, via disconnect(). Delete the corresponding Connection.

7. Delete the UserEnvironment.

The Datastore_CLI connection operations are:

```
void connect (in Connection connection,
              in string datastore_id,
              in string user_name,
              in string authentication);
```

This opens a connection to the Datastore using the Connection. A client can establish more than one connection, but only one connection can be current at a time. The connection that connect() establishes becomes the current connection.

```
void disconnect (in Connection connection);
```

This closes the Connection.

```
Connection get_connection (
              in string datastore_id,
              in string user_name);
```

This returns the Connection associated with the datastore_id.

When any of the data manipulation operations is called, a datastore transaction begins implicitly if the Connection involved is not already active. A Connection becomes active once the transaction begins and remains active until transact() is called.

The Datastore_CLI data manipulation operations are:

```
void add_object (in Connection connection,
              in CosPersistenceDDO::DDO data_obj);
```

This adds the DDO to the Datastore. If necessary, get the mapping schema information for the DDO first.

```
void delete_object (in Connection connection,
              in CosPersistenceDDO::DDO data_obj);
```

This deletes the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first.

```
void update_object (in Connection connection,
              in CosPersistenceDDO::DDO data_obj);
```

This updates the DDO in the Datastore. If necessary, get the mapping schema information for the DDO first.

```
void retrieve_object (in Connection connection,
              in CosPersistenceDDO::DDO data_obj);
```

This retrieves the DDO from the Datastore. If necessary, get the mapping schema information for the DDO first. To improve performance, the DBDatastore_CLI may obtain access to more than one DDO at a time and cache these.

```
Cursor select_object (in Connection connection,
```

in string key);

This selects and retrieve the DDO(s) which match the key from the Datastore. The DDO(s) are returned through the Cursor. If necessary, get the mapping schema information for the key first. This operation is provided to support the Query Service. In addition, the Datastore_CLI will support any other operation required by the Object Query Service.

The Datastore_CLI functions as a resource manager for the DDOs that it manages. As such, it will support all resource manager operations specified by the Transaction Service. When the Transaction Service is not being used, a transaction is initiated implicitly by either a Connection or a transact(), and ended with a transact():

void transact (in UserEnvironment user_envir,
            in short completion_type);

This completes (commit or rollback) a Datastore transaction. Transaction completion enacts or undoes any add_object(), update_object() or delete_object() operations performed on any Connection within the UserEnvironment since the connection was established or since a previous call to transact() for the same UserEnvironment. The values of completion_type are specified in the X/Open CLI Specification.

The Datastore_CLI PID Operations are:

void assign_PID (in PID_CLI p);

This assign a value for the id attribute of the pid. The first attribute, datastore_type, must be filled in before calling this operation. If only the first attribute is filled in, then this operation will fill in the second attribute, datastore_id, as well.

void assign_PID_relative (in PID_CLI source_pid,
            in PID_CLI target_pid);

This assigns values for the attributes of the target_pid based on the values of the source_pid. The target_pid's first two attributes, datastore_type and datastore_id, will be assigned the same values as those of the source_pid. Its id attribute will be assigned a new value which is based on some relationship with that of the source_pid. The algorithm defining that relationship is up to the implementation.

boolean is_identical_PID (in PID_CLI pid_1, in PID_CLI pid_2);

This tests to see if the two pids are identical. In order for the two pids to be identical, the following conditions must be true:

1. Both pids must be managed by this PDS
2. all three attributes of the pids must be identical individually.

string get_object_type (in PID_CLI p);

This gets the object_type of the pid.

Other Datastore_CLI operations are:

void register_mapping_schema (in string schema_file);
This registers the mapping schema information contained within the schema_file with the Datastore_CLI. The mapping schema generally consist of individual mappings each of which is applicable to a given pair of object_type and datastore_type.

Cursor execute (in Connection connection,
in string command);
This executes a command on the Datastore. If there are any DDOs to be returned as a result, this is done through the Cursor.

5.16 Other Datastores

There are other Datastore interfaces that can be used by PDSs. Some of these interfaces are not CORBA object interfaces, in that they are not defined in IDL and the Datastores are not objects.

Some Datastores are simple, such as POSIX files. Others may be databases, and may use generic interfaces for databases and record files such as SQL, the X/Open CLI API, IDAPI or ODBC. Some Datastores are tuned to support nested documents or other specific kinds of objects such as Bento.

Because the Datastore interface is not exposed to object implementations or clients, the choice of Datastore interface is up to the PDS. So long as the PDS can support its Protocol using the particular Datastore interface, any implementation of the Datastore can be used by that PDS. The identification of data within different types of Datastores is facilitated by the PID, which can be specialized to each Datastore type.

5.17 Standards Conformance

This service is specified in standard OMG IDL.

The Datastore_CLI portion of the Persistent Object Service is consistent with the X/Open CLI draft standard.

The ODMG-93 PDS Object Protocol incorporates the ODMG-93 specificiation.

5.18 References

The X/Open CLI standard is documented in *X/Open Data Management Call Level Interface (CLI) Draft Preliminary Specification*. Reading, UK: X/Open Ltd., 1993.

The IDAPI standard is documented in *IDAPI Working Draft*. Scotts Valley, CA: Borland International, August 1993.

The term “ODBC” refers to *Microsoft Open Database Connectivity Software Development Kit, Programmer Reference*, Version 1.0. Redmond, WA: Microsoft Corp., 1992.

The term “Bento” refers to Jed Harris and Ira Rubin, *The Bento Specification, Revision 1.0d5*. Cupertino, CA: Apple Computer, Inc., July 15, 1993,

The term “ODMG-93” refers to R.G.G.Cattell, T.Atwood, J.Duhl, G.Ferran, M.Loomis, and D.Wade, *The Object Database Standard: ODMG-93*. San Mateo, CA: Morgan Kaufmann, 1993.

Life Cycle Service Specification 6

6.1 Service Description

6.1.1 Overview

Life Cycle Service defines services and conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, the Life Cycle Service defines conventions that allow clients to perform life cycle operations on objects in different locations.

This overview describes the life cycle problem for distributed object systems.

The problem of creation

Figure 6-1 illustrates the problem of a client in one location creating an object in another.

Figure 6-1 Life Cycle service defines how a client can create an object “over there”.

To create an object in a different location, the following questions must be answered:

- Can the client control the location for the new object?

- On the other hand, can the location be determined according to some administered policy?
- What entity does the client communicate with in order that a new object is created?
- How does the client find that entity?
- How much control does the client have over deciding the implementation of the created object?
- Can the client influence the initial values of the newly created object?
- Can the client create an object in an implementation specific fashion?

The problem of moving or copying an object

Figure 6-2 illustrates the problem of moving or copying an object in a distributed object system.

Figure 6-2 Life Cycle Service defines how a client can move or copy an object over there.

To support moving or copying an object, the following questions must be answered:

- Can the client control the location for the copied or migrated object?
- On the other hand, can the location be determined according to some administered policy?
- What entity does the client communicate with to copy or migrate the object?
- How does the client find that entity?
- What happens to the implementation code of a copied or migrated object?

The problem of operating on a graph of distributed objects

Distributed objects do not *float* in space; they are connected to one another. The connections are called *relationships*. Relationships allow semantics to be added to references between objects. For example, relationships allow one object to *contain* another. Life Cycle services must work in the presence of graphs of related objects.

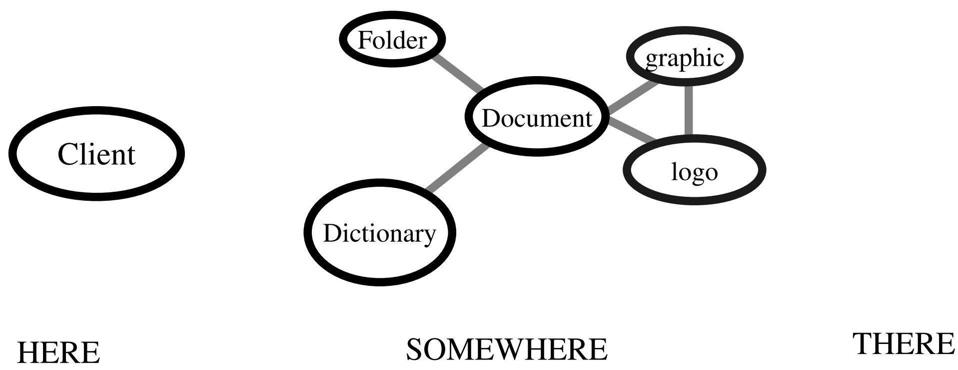


Figure 6-3 The object life cycle problem for graphs of objects is to determine the boundaries of a graph of objects and operate on that graph. In the above example, a document contains a graphic and a logo, refers to a dictionary and is contained in a folder.

Figure 6-3 illustrates the object life cycle problem for graphs of objects. In the example, the folder *contains* a document, the document *contains* a graphic and a logo and *references* a dictionary. The graphic *references* the logo that is *contained* in the document. For graphs of objects, life cycle services must answer the following questions:

- What are the boundaries of the graph? For example, if a client copies the document, which objects are affected?
- If multiple objects are affected, how is the life cycle operation actually applied to those objects?
- Are cycles in the graph preserved? For example, if copying the document results in copying the graphic and the logo, is the cycle preserved in the copy?

6.1.2 Organization of this Chapter

This specification defines services and conventions to answer these life cycle issues.

Section 6.1.3 specifies a *client's model* of object life cycle. It describe the model a client has of factories and life cycle operations. A wide variety of implementations of this model are possible.

Section 6.1.4 discusses *factory finders* in detail.

Section 6.2 defines the *CosLifeCycle* module. This module defines the service interfaces and the interface supported by objects that participate in the service.

Section 6.3 discusses factory implementation strategies.

Section 6.4 discusses how objects can use factories and factory finders to support the *copy* and *move* operations.

Section 6.5 summarizes the object life cycle framework.

Appendix A contains an addendum to the Life Cycle Service; the addendum provides a specification for compound life cycle operations.

This chapter also includes additional appendices that are not part of the Life Cycle Service specification: they are included as background material. Appendix B suggests a filtering language for the filter criteria. Appendix C discusses administration of generic factories. Appendix D discusses support for PCTE objects.

6.1.3 Client's Model of Object Life Cycle

A client is any piece of code that initiates a life cycle operation for some object. A client has a simple view of the life cycle operations.

Client's Model of Creation

The client's model of creation is defined in terms of factory objects. A factory is an object that creates another object. Factories are *not* special objects. As with any object, factories have well-defined IDL interfaces and implementations in some programming language.

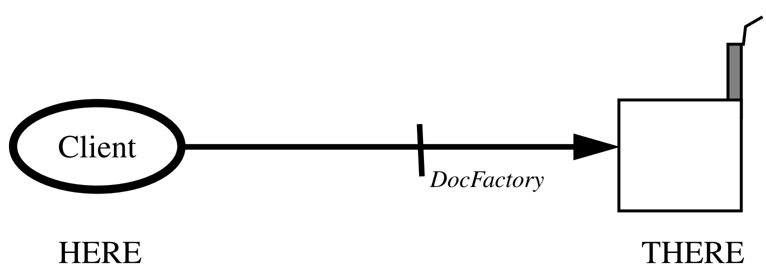


Figure 6-4 To create an object "over there" a client must posses an object reference to a factory over there. The client simply issues a request on the factory.

There is no standard interface for a factory. Factories provide the client with specialized operations to create and initialize new instances in a natural way for the implementation. Figure 6-5 illustrates a factory for a document.

```
interface DocFactory {
    Document create();
    Document create_with_title(in string title);
```

```
        Document create_for(in natural_language nl);
    };
```

Figure 6-5 An example of a document factory interface. This interface is defined for clients as a part of application development.

Factories are object *implementation* dependent. A different implementation of the document could define a different factory interface.

While there is no standard interface for a factory, a *generic factory* interface is defined by the life cycle service in section 6.2.3. A generic factory is a creation service. It provides a generic operation for creation. Instead of invoking an object specific operation on a factory with statically defined parameters, the client invokes a standard operation whose parameters can include information about resource filters, state initialization, policy preferences, etc.

To create an object, a client must possess an object reference for a factory, which may be either a generic factory or an object-specific factory, and issue an appropriate request on the factory. As a result, a new object is created and typically an object reference is returned.

There is nothing special about this interaction.

A factory assembles the resources necessary for the existence of an object it creates. Therefore, the factory represents a scope of resource allocation, which is the set of resources available to the factory. A factory may support an interface that enables its clients to constrain the scope.

Clients find factory objects in the same fashion they find any object. Two common scenarios for clients to find factories are:

- Clients use a finding mechanism, such a naming context, drag-and-drop, or a trader, to find factories.
- Clients are passed factory objects as a parameter to an operation the client supports.

Various *implementation* strategies for factories are discussed in detail in section 6.3.

Client's Model of Deleting an Object

A client that wishes to delete an object issues a remove[1] request on an object supporting the *LifeCycleObject* interface. (The *LifeCycleObject* interface is defined in section 6.2.) The object receiving the request is called the *target.*

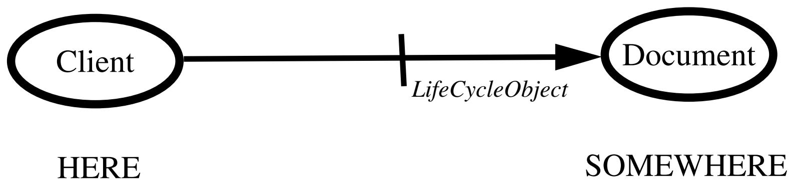


Figure 6-6 To delete an object, a client must posses an object reference supporting the *LifeCycleObject* interface and issues a remove request on the object.

Figure 6-6 illustrates a client deleting the document.

Client's Model of Copying or Moving an Object

A client that wishes to move or copy an object issues a move or copy request on an object supporting the *LifeCycleObject* interface. The object receiving the request is called the *target.*

The move and copy operations expect an object reference supporting the *FactoryFinder* interface. The factory finder represents the "THERE" in Figure 6-7. The client is indicating to move or copy the target using a factory within the scope of the factory finder. Section 6.1.4 describes factory finders in more detail.

1. The operation is named remove, rather than delete, because delete collides with the delete operator in C++.

The implementations of move and copy can use the factory finder to find appropriate factories “over there”. Section 6.4 describes how objects can implement move and copy using the factory finder. This is invisible to the client.

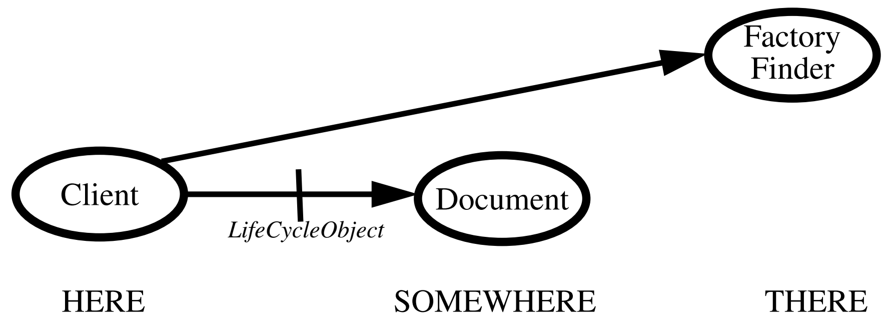


Figure 6-7 Life cycle services define how a client can move or copy an object from here to there.

In the example of Figure 6-7, client code would simply issue a copy request on the document and pass it an object supporting the *FactoryFinder* interface as an argument.

When a client issues a copy request on a target, it is assumed that the target, the factory finder, and the newly created object can all communicate via the ORB. With externalization/internalization there is no such assumption. In the presence of a future externalization service, the externalized form of the object can exist outside of the ORB for arbitrary amounts of time, be transported by means outside of the ORB and can be internalized in a different, disconnected ORB.

Note – In general, a client is unaware of how a target and a factory finder are implemented. The target may represent a simple object or it may represent a graph of objects. Similarly, a factory finder may represent a very concrete location, such as a specific storage device, or it may represent a more abstract location, such as a group of machines. The client uses the same interface in all of these cases.

6.1.4 Factory Finders

Factory finders support an operation, find_factories, which returns a sequence of factories. Clients pass factory finders to the move and copy operations, which typically invoke this operation to find a factory to interact with. (This is described in detail in section 6.4.) The new copy or the migrated object will then be within the scope of the factory finder.

Some examples of locations that a factory finder might represent are:

- somewhere on a work group’s local area network
- storage device A on machine X
- Susan’s notebook computer

Multiple Factory Finders

The factory finder interface given in section 6.2 represents the minimal functionality supported by all factory finders. Target implementations can depend on this operation being available. More sophisticated factory finding facilities can be provided by extended finding services.

Currently, the only finding service being considered for standardization by the OMG is the naming service. Others are likely to be standardized in the future. It is likely that there will always be multiple finding services, of different expressive powers, in distributed object systems.

As demonstrated in Figure 6-8, the *FactoryFinder* interface can be mixed-in with interfaces for finding services, allowing multiple finding services. Many clients simply pass factory finders on to target objects. However, objects that need the services of a more powerful finding mechanism can narrow the factory finder to an appropriate, more specific interface.

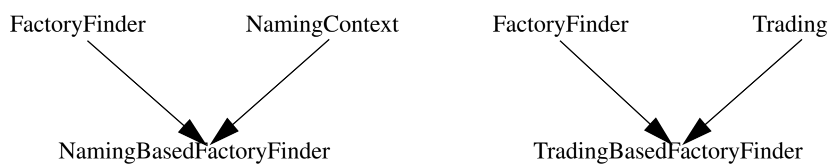


Figure 6-8 The *FactoryFinder* interface can be “mixed in” with interfaces of more powerful finding services.

The power of a factory finder is determined by the power of the finding service.

6.1.5 Design Principles

Several principles have driven the design of the Life Cycle Service:

1. A factory object registered at a factory finder represents an implementation at that location. Thus, a factory finder allows clients to query a location for an implementation.
2. Object implementations can embody knowledge of finding a factory, relative to a location. Object implementations usually do not embody knowledge of location.
3. The desired result for life cycle operations such as copy and move depends on relationships between the target object and other objects. The design given in Appendix A has built-in support for the two most basic kinds of relationships, *containment* and *reference*, and supports the definition of new kinds of relationships and propagation semantics.
4. The Life Cycle Service is not dependent on any particular model of persistence and is suitable for distributed, heterogeneous environments.
5. The design does not include an object equivalence service nor rely on global object identifiers.

6.1.6 Resolution of Technical Issues

This specification addresses the following issues that were identified for the Life Cycle Service in the OMG *Object Services Architecture*[2] :

- *Creation*: Many of the parameters supplied to an object create operator will be implementation-dependent, so that a standardized universal IDL signature for object creation is not possible. IDL signatures for object creation will be defined for various kinds of object factories, but the signatures will be specific to type, implementation, and persistent storage mechanism of the object to be created.
- *Deletion*: A remove operator is defined on any object supporting the *LifeCycleObject* interface. This model for deletion supports any desired paradigm for referential integrity. Appendix A describes support for the two most common paradigms, based on reference and containment relationships. Only one type of deletion is supported; a different operation should be used for archiving an object. This interface can support many paradigms for storage management, e.g. garbage collection and reference counts. Since storage management is implementation-dependent, its interface does not belong in the generalized life cycle interfaces.
- *Copying*: Appendix A describes support for shallow and deep copy, and referential integrity. A scheme based on reference and containment relationships defines scopes for operations such as copy. The concept of an *factory finder* is used for object location. This paradigm for copying, deleting, and moving objects works regardless of an object's ORB, persistent storage mechanism, and implementation. This design is extensible because objects participate in the traversal algorithm, and the relationship service presented in the appendix supports the definition of new kinds of relationships with different behavior.
- *Equivalence*: There was no need for an object equivalence service or global object identifiers in the design of the Life Cycle Service to support real world applications or other object services.

2. *Object Services Architecture*, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

6.2 The CosLifeCycle Module

Client code accesses the basic life cycle functionality via the *CosLifeCycle* module. This module defines the *FactoryFinder, LifeCycleObject* and *GenericFactory* interfaces and describes the operations of these interfaces in detail.

```
#include "Naming.idl"

module CosLifeCycle{

	typedef Naming::Name Key;
	typedef Object Factory;
	typedef sequence <Factory> Factories;
	typedef struct NVP {
		Naming::Istring name;
		any   value;
	} NameValuePair;
	typedef sequence <NameValuePair> Criteria;

	exception NoFactory {
		Key search_key;
	};
	exception NotCopyable { string reason; };
	exception NotMovable { string reason; };
	exception NotRemovable { string reason; };
	exception InvalidCriteria{
		Criteria invalid_criteria;
	};
	exception CannotMeetCriteria {
		Criteria unmet_criteria;
	};
```

Figure 6-9 The CosLifeCycle Module

```
        interface FactoryFinder {
                Factories find_factories(in Key factory_key)
                        raises(NoFactory);
        };

        interface LifeCycleObject {
                LifeCycleObject copy(in FactoryFinder there,
                                in Criteria the_criteria)
                        raises(NoFactory, NotCopyable, InvalidCriteria,
                                 CannotMeetCriteria);
                void move(in FactoryFinder there,
                                in Criteria the_criteria)
                        raises(NoFactory, NotMovable, InvalidCriteria,
                                 CannotMeetCriteria);
                void remove()
                        raises(NotRemovable);
        };

        interface GenericFactory {
                boolean supports(in Key k);
                Object create_object(
                                in Key          k,
                                in Criteria   the_criteria)
                        raises (NoFactory, InvalidCriteria,
                                CannotMeetCriteria);
        };
};
```

Figure 6-9 The CosLifeCycle Module

6.2.1 The LifeCycleObject Interface

The *LifeCycleObject* interface defines copy, move and remove operations. Objects participate in the life cycle service by supporting this interface.

copy

```
        LifeCycleObject copy(in FactoryFinder there,
                        in Criteria the_criteria)
                raises(NoFactory, NotCopyable, InvalidCriteria,
                        CannotMeetCriteria);
```

The copy operation makes a copy of the object. The copy is located in the scope of the factory finder passed as the first parameter. The copy operation returns an object reference to the new object. The new object is initialized from the existing object.

The first parameter, there, may be a nil object reference. If passed a nil object reference, the target object can determine the location or fail with the NoFactory exception.

The second parameter, the_criteria, allows for a number of optional parameters to be passed. Typically, the target simply passes this parameter to the factory used in creating the new object. The criteria parameter is explained in detail in section 6.2.4

If the target cannot find an appropriate factory to create a copy "over there", the NoFactory exception is raised. An implementation that refuses to copy itself should raise the NotCopyable exception. If the target does not understand the criteria, the InvalidCriteria exception is raised. If the target understands the criteria but cannot satisfy the criteria, the CannotMeetCriteria exception is raised.

In addition to these exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied object, NO_RESOURCES will be raised. Similarly, if a target does not implement the copy operation, the NO_IMPLEMENT exception will be raised.

It is implementation dependent whether this operation is *atomic*.

move

```
void move(in FactoryFinder there,
          in Criteria the_criteria)
     raises(NoFactory, NotMovable, InvalidCriteria,
          CannotMeetCriteria);
```

The move operation on the target moves the object to the scope of the factory finder passed as the first parameter. The object reference for the target object remains valid after move has successfully executed.

The first parameter, there, may be a nil object reference. If passed a nil object reference, the target object can determine the location or fail with the NoFactory exception.

The second parameter, the_criteria, allows for a number of optional parameters to be passed. Typically, the target simply passes this parameter to the factory used in migrating the new object. The criteria parameter is explained in detail in section 6.2.4

If the target cannot find an appropriate factory to support migration of the object "over there", the NoFactory exception is raised. An implementation that refuses to move itself should raise the NotMovable exception. If the target does not understand the criteria, the InvalidCriteria exception is raised. If the target understands the criteria but cannot satisfy the criteria, the CannotMeetCriteria exception is raised.

In addition to these exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for migrating the object, NO_RESOURCES will be raised. Similarly, if a target does not implement the move operation, the NO_IMPLEMENT exception will be raised.

It is implementation dependent whether this operation is *atomic*.

remove

```
void remove()
      raises(NotRemovable);
```

Remove instructs the object to cease to exist. The object reference for the target is no longer valid after remove successfully completes. The client is not responsible for cleaning up any resources the object uses. An implementation that refuses to remove itself should raise the NotRemovable exception. In addition to this exception, implementations may raise standard CORBA exceptions.

6.2.2 The FactoryFinder Interface

Factory finders support an operation, find_factories, which returns a sequence of factories. Clients pass factory finders to the move and copy operations, which typically invoke this operation to find a factory to interact with. (This is described in detail in section 6.4.)

The factory finder interface represents the *minimal* functionality supported by all factory finders.

find_factories

```
Factories find_factories(in Key factory_key)
      raises(NoFactory);
```

The find_factories operation is passed a key used to identify the desired factory. The key is a name, as defined by the naming service. More than one factory may match the key. As such, the factory finder returns a sequence of factories. If there are no matches, the NoFactory exception is raised.

The scope of the key is the factory finder. The factory finder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factories or objects they create.

It is beyond the scope of this specification to standardize the key space. The space of keys is established by *convention* in particular environments. The *kind* field[3] of the key is useful for partitioning the key space. Suggested values for the id and kind fields are given in Table 6-1.

Table 6-1 Suggested conventions for factory finder keys.

id field	**kind field**	**meaning**
name of object interface	"object interface"	Find factories that create objects supporting the named interface.
name of equivalent implementations	"implementation equivalence class"	Find factories that create objects with implementations in a named equivalence class of implementations.[1]
name of object implementation	"object implementation"	Find factories that create objects of a particular implementation.
name of factory interface	"factory interface"	Find factories supporting the named factory interface.

1. An example of an implementation equivalence class is a set of object implementations that have compatible externalized forms.

6.2.3 The GenericFactory Interface

In many environments, management of a set of resources that are allocated to objects at creation time is required. This needs to be done in a coordinated fashion for all types of objects. The Life Cycle Service provides a framework for this which is intended to be usable in a variety of administrative environments. However, the differing environments will administer a variety of resources and it is beyond the scope of this framework to identify all the possible types of resource.

While there is no standard interface for a factory, a *GenericFactory* interface is defined. The *GenericFactory* interface defines a generic creation operation, create_object. By defining a generic interface for creation, a creation service can be implemented. This is particularly useful in environments where administering a set of resources is important.

Such a generic factory can implement resource policies and represent multiple locations. In administered environments, object specific factories, such as the document factory described in section , may delegate the creation process to the generic factory. This is described in detail in section 6.3.2.

The job of the generic factory is to match the creation criteria specified by clients of the *GenericFactory* interface with offers made on behalf of implementation specific factories.

3. See the naming service specification.

Figure 6-10 illustrates the structure of a creation service.

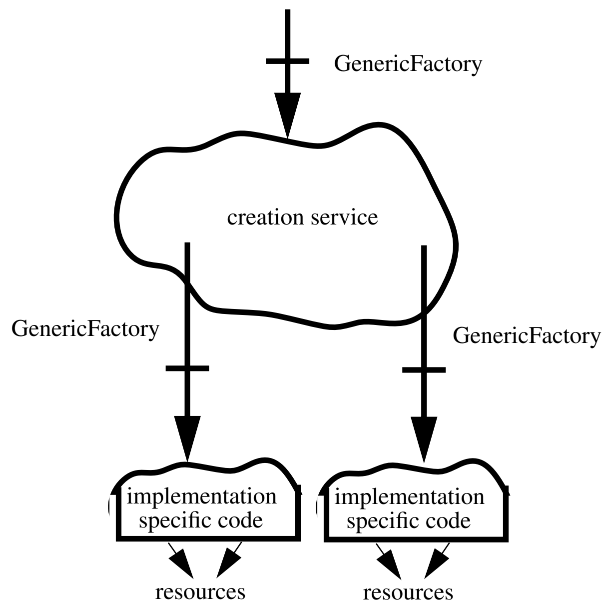


Figure 6-10 The Life Cycle service provides a generic creation capability. Ultimately, implementation specific creation code is invoked by the creation service. The implementation specific code also supports the *GenericFactory* interface.

The client of the *GenericFactory* interface invokes the create_object operation and can express criteria for creation.

Ultimately, this request will be passed to an implementation specific factory which supports the *GenericFactory* interface. To get there, the request may travel through a number of generic factories. However, all of this is transparent to the client.

create_object

```
Object create_object(
        in Key k,
        in Criteria the_criteria)
    raises (NoFactory, InvalidCriteria,
        CannotMeetCriteria);
```

The create_object operation is passed a key used to identify the desired object to be created. The key is a name, as defined by the Naming Service.

The scope of the key is the generic factory. The generic factory assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the created object.

It is beyond the scope of this specification to standardize the key space. The space of keys is established by *convention* in particular environments. The *kind* field[4] of the key is useful for partitioning the key space. *Suggested* values for the id and kind fields are given in Table 6-2.

Table 6-2 Suggested conventions for generic factory keys.

id field	**kind field**	**meaning**
name of object interface	"object interface"	Create an object that supports the named interface.
name of equivalent implementations	"implementation equivalence class"	Create an object whose implementation is in a named equivalence class of implementations.[1]
name of object implementation	"object implementation"	Create objects of a particular implementation.

1. An example of an implementation equivalence class is a set of object implementations that have compatible externalized forms

The second parameter, the_criteria, allows for a number of optional parameters to be passed. Criteria are explained in detail in section 6.2.4

If the generic factory cannot create an object specified by the key, then NoFactory is raised.

If the target does not understand the criteria, the InvalidCriteria exception is raised. If the target understands the criteria but cannot satisfy the criteria, the CannotMeetCriteria exception is raised.

supports

```
boolean supports(in Key k);
```

The supports operation returns true if the generic factory can create an object, given the key. Otherwise false is returned.

4. See the naming service specification.

6.2.4 Criteria

The create_object operation of the *GenericFactory* interface expects a parameter specifying the creation criteria. The move and copy operations of the *LifeCycleObject* interface also expects this parameter; typically they pass it through to a factory. This section documents this parameter.

The criteria parameter is expressed as an IDL sequence of name-value pairs. In particular, it is described by the following data structure given in the *CosLifeCycle* module:

```
typedef struct NVP {
        Naming::Istring name;
        any value;
} NameValuePair;
typedef sequence <NameValuePair> Criteria;
```

The parameter is given as a sequence of name-value pairs in order to be extensible and support “pass-through”; that is, new name-value pairs can be defined in the future and objects can be written that do not interpret the name-value pairs, but just pass them on to other objects.

Note – It is beyond the scope of this specification to standardize particular criteria. Supporting criteria is optional. Furthermore, supporting different criteria is acceptable. The criteria given here are suggestions.

Table 6-3 suggests criteria to be supported by the generic factory. Detailed descriptions follow.

Table 6-3 Suggested criteria.

criterion name	**type of criterion value**	**interpretation**
“initialization”	sequence<NameValuePair>	initialization parameters, given as a sequence of name-value pairs.
“filter”	string	allows clients of the generic factory to express a constraint on the created object.
“logical location”	sequence<NameValuePair>	allows clients of the generic factory to express a connection for the object, for example a PCTE relationship.
“preferences”	string	a way for clients to influence the policies that a generic factory may use when creating an object

"initialization"

The "initialization" criterion is a sequence of name-value pairs which is intended to contain application specific initialization values. Typically, the generic factory will pay no attention to the initialization criterion and simply passes it on to application specific factory code.

"filter"

The filter criterion is a constraint expression which provides the client with a powerful way of expressing its requirements on creation. The generic factory will use the constraint expression to make decisions about the allocation of particular resources. For example, a client could give a constraint "operating system" != "windows nt".

These constraints are expressed in some Constraint Language. A constraint language is suggested in Appendix B.

Filters are potentially complex and InvalidCriteria will be raised if the filter is too complex for the factory or is syntactically incorrect.

"logical location"

The "logical location" criterion allows a client to express where a created/copied/migrated object is logically created. For example, in PCTE an object is always in a relationship with another object. In such an environment, the logical location would specify another object and a relationship.

"preferences"

The "preferences" criterion allows the client to influence the policies which the generic factory uses to make decisions. For example, a generic factory might arbitrarily choose a machine from a set of machines. Using the preferences criterion, a client could express its preference for a particular machine. Policies and preferences are described in more detail in Appendix B.

6.3 Implementing Factories

As defined under Client's Model of Creation on page 4, any object that creates another object in response to some request is called a *factory*. Clients depend only on the definitions in that section.

The client's model of object life cycle has intentionally been defined abstractly. This allows a wide variety of implementation strategies.

Factories are *not* special objects. They have well-defined IDL interfaces and implementations in programming languages. Defining factory interfaces and implementing them are a normal part of application development.

Ultimately, the creation process requires implementation dependent code that assembles resources for the storage and execution of an object. The act of creating an object requires assembling and initializing all of the resources required to support the execution and storage of the object. The resources typically include:

- the allocation of one or more BOA object references, and
- resources related to persistence storage.

6.3.1 Minimal Factories

Figure 6-11 illustrates a minimal implementation of a factory that assembles resources in a single factory object.

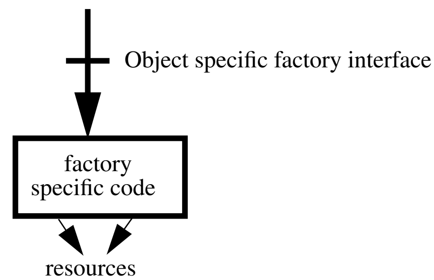


Figure 6-11 Factories assemble resources for the execution of an object. A minimal implementation achieves this with a single factory implementation.

6.3.2 Administered Factories

Factories can delegate the creation process to a generic factory that administers a set of resources. The generic factory may apply policies to all creation requests.

Eventually such a generic creation service, needs to communicate with implementation specific code that actually assembles the resources for the object. Figure 6-12 illustrates an object specific factory, such as the document factory of Figure 6-5 that delegates the creation problem to the generic creation service. The object-specific factory effectively adds a statically typed wrapper around the generic factory.

.

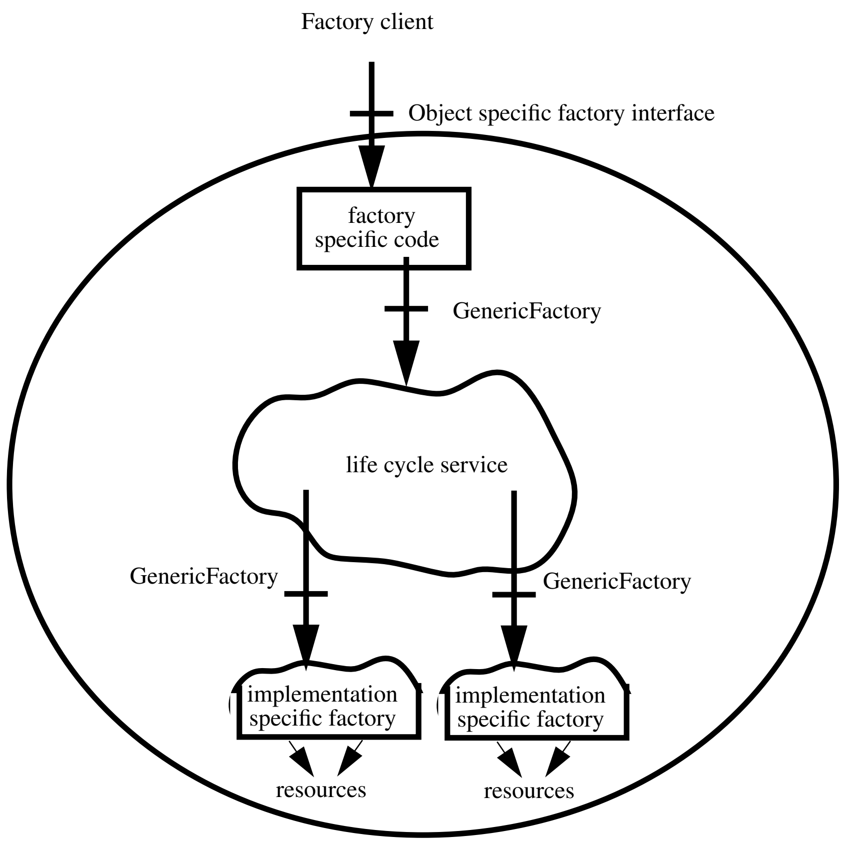


Figure 6-12 In an administered environment, factory *implementations* can delegate the creation problem to a generic factory. The generic factory can apply resource allocation policies. Ultimately the creation service communicates with implementation specific code that assembles resources for the object.

6.4 Target's Use of Factories and Factory Finders

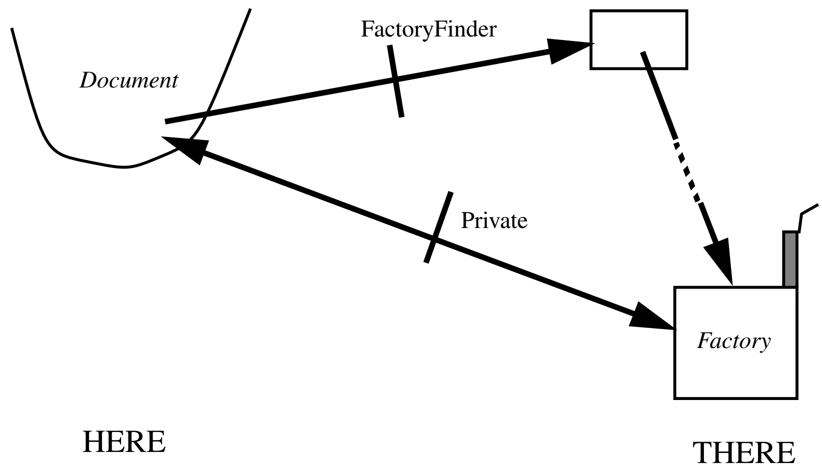


Figure 6-13 The copy and move operations are passed a *FactoryFinder* to represent "there." The implementation of the target uses the *FactoryFinder* to find a factory object for creation over there. The protocol between the object and the factory is private. They can communicate and transfer state according to any implementation-defined protocol.

A client passes a factory finder as a parameter to a copy or move request.

Clients do not generally understand the implementation constraints of the object being copied. Clients cannot express what the target object needs in order to copy itself to the new location.

Target object implementations, on the other hand, put constraints on factories based on implementation concerns. It is unlikely that target implementation code is interested in further constraining location.

To find an appropriate factory, the target object implementation may use the factory finder with its minimal interface defined in section 6.2.2 or it may attempt to narrow the factory finder to a more sophisticated finding service with more expressive power. The target object implementation can always depend on the existence of the minimal interface.

Once the target object implementation finds a factory, it communicates with the factory using a private, implementation-defined, interface.

6.5 Summary of Life Cycle Service

The problem of distributed object life cycle is the problem of

- Creating an object
- Deleting an object

- Moving and copying an object
- Operating on a graph of distributed objects.

The client's model of object life cycle is based on *factories* and target objects supporting the *LifeCycleObject* interface. Factories are objects that create other objects. The *LifeCycleObject* interface defines operations to delete an object, to move an object and to copy an object.

A *GenericFactory* interface is defined. The generic factory interface is sufficient to create objects of different types. By defining a *GenericFactory* interface, implementations that administer resources are enabled.

6.5.1 Summary of Life Cycle Service Structure

The Life Cycle Service specification consists of these interfaces:

- *LifeCycleObject*
- *FactoryFinder*
- *GenericFactory*
- Interfaces described in Appendix A, an addendum to the Life Cycle Service

Appendix A Addendum to Life Cycle Service: Compound Life Cycle Specification

This appendix contains the specification for the compound life cycle component of the Life Cycle Service .The compound life cycle specification depends on the Life Cycle Service for the definition of the client view of Life Cycle operations. Moreover, it extends the Life Cycle Service to support compound life cycle operations on graphs of related objects. In addition, the compound life cycle specification depends on the Relationship Service for the definition of object graphs.

The Life Cycle Service specification describes a client's view of object life cycle. It describes how a client can *c*reate, copy, move and remove objects in a distributed object system. To create objects, clients find *factory objects* and issue create requests on factories. To copy, move and remove objects, clients issue requests on target objects supporting the *LifeCycleObject* interface.

If the target object represents a simple object, that is an object that is not part of a graph of related objects, the target provides an implementation for each of the operations in the *LifeCycleObject* interface.

If, on the other hand, the target object uses the Relationship Service for representing relationships with other objects, additional services are available to implement the compound life cycle operations. The specification in this appendix describes those services.

A.1 Key Features

The compound life cycle specification:

- Addresses the issues of copying, moving and removing objects that are related to other objects. Depending on the semantics of the relationships, these life cycle operations are applied to:
- the object, to the relationship and to the related objects
- the object and to the relationship
- the object
- Coordinates compound life cycle operations on graphs of related objects, thus relieving object developers from implementing compound operations.
- Illustrates a general model for applying compound operations to graphs of related objects. The Externalization Service also illustrates the model.

A.2 Service Structure

The specification in this appendix defines a service that applies a compound life cycle operation to a graph of related objects, given a starting node. Compound operations traverse a graph of related objects and apply the operation to the relevant nodes, roles and relationships of the graph. The service supports the *CosCompoundLifeCycle::Operations* interface. Implementations of the service depend on the *CosCompoundLifeCycle::Node* , *CosCompoundLifeCycle::Role* and *CosCompoundLifeCycle::Relationship* interfaces which are subtypes of the *Node* , *Role* and *Relationship* interfaces defined in the Relationship Service. The *CosCompoundLifeCycle::Node* , *CosCompoundLifeCycle::Role* and *CosCompoundLifeCycle::Relationship* interfaces add operations to copy, remove and move nodes, roles and relationships.

The Relationship Service defines interfaces for containment and reference relationships and their roles. This appendix defines interfaces that inherit those interfaces and the compound life cycle interfaces.

A.3 Interface Overview

Table 6-4 and Table 6-5 summarize the interfaces defined in the *CosCompoundLifeCycle* module. The *CosCompoundLifeCycle* module is described in detail in sectionSection A.4.2.

Table 6-4 Interfaces defined in the *CosCompoundLifeCycle* module for initiating compound life cycle operations.

Interface	Purpose
Operations	Defines compound life cycle operations on graphs of related objects.
OperationsFactory	Defines an operation to create an object that supports the *Operations* interface.

Table 6-5 Interfaces defined in the *CosCompoundLifeCycle* module that are used by implementations of compound life cycle operations

Interface	Inherits	Purpose
Node	CosGraphs::Node	Defines life cycle operations on nodes in graphs of related objects.
Relationship	CosRelationships::Relationship	Defines life cycle operations on relationships.
Role	CosGraphs::Role	Defines life cycle operations on roles.
PropagationCriteriaFactory		Creates an object that supports the CosGraphs::TraversalCriteria interface that uses relationship propagation values.

Table 6-6 and Table 6-7 summarize the interfaces that combine the specific relationships defined by the Relationship Service and the life cycle interfaces defined in this appendix.

Table 6-6 Interfaces defined in the *CosLifeCycleContainment* module.

Interface	Inherits	Purpose
Relationship	CosContainment::Containment and CosCompoundLifeCycle::Relationship	Combines both interfaces. No additional operations are defined.
ContainsRole	CosContainment::ContainsRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.
ContainedInRole	CosContainment::ContainedInRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.

Table 6-7 Interfaces defined in the *CosLifeCycleReference* module.

Interface	Inherits	Purpose
Relationship	CosContainment::Reference and CosCompoundLifeCycle::Relationship	Combines both interfaces. No additional operations are defined.
ReferencesRole	CosContainment::ReferencesRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.
ReferencedByRole	CosContainment::ReferencedByRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.

A.4 Compound Life Cycle Operations

The Life Cycle specification describes a client's view of object life cycle. It describes how a client can *create*, *copy*, *move* and *remove* objects in a distributed object system. To create objects, clients find *factory objects* and issue create requests on factories. To copy, move and remove objects, clients issue requests on target objects supporting the *LifeCycleObject* interface.

If the target object represents a simple object, that is an object that is not part of a *graph of related* objects, the target provides an implementation for each of the operations in the *LifeCycleObject* interface.

If the target participates as a node in a graph of related objects, the target can delegate the life cycle operation to a service that implements the compound life cycle operation. In particular, the target simply creates an object that supports the *CosCompoundLifeCycle::Operations* interface and issues the corresponding life cycle request on it. The compound life cycle operations expect a *CompoundLifeCycle::Node* object reference as a starting node. The target simply passes its *CompoundLifeCycle::Node* object reference as the starting node.

When the life cycle object has completed issuing compound life cycle requests, it simply issues the destroy request to destroy the compound operation.

Figure 6-14 illustrates the target's delegation of the life cycle request to compound operation.

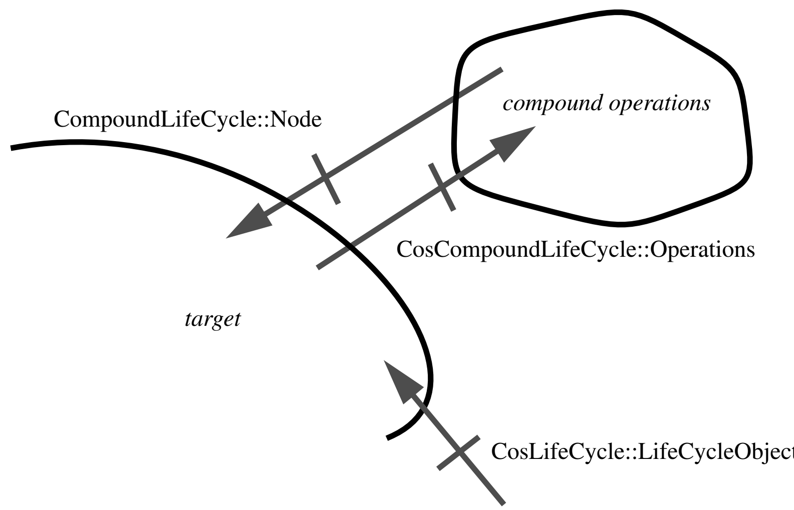


Figure 6-14 A life cycle object that is part of a graph of related objects delegates the orderly operation on the graph to an object that implements the compound life cycle operation.

A.4.1 Applying the Copy Operation to the Example

We now use the example in the Relationship Service Specification (Figure 9-3 on page 9-9) to illustrate applying the copy operation to a graph. Figure 6-15 illustrates the graph and the compound operation prior to applying the copy operation. Recall that the folder *contains* the document; the document is *contained in* the folder. The document *contains* the figure; the figure is *contained in* the document. The document *contains* the logo and the logo is *contained in* the document. On the other hand, the document *references* the book; the book is *referenced by* the document. Finally, the figure *references* the logo; the logo is *referenced by* the figure.

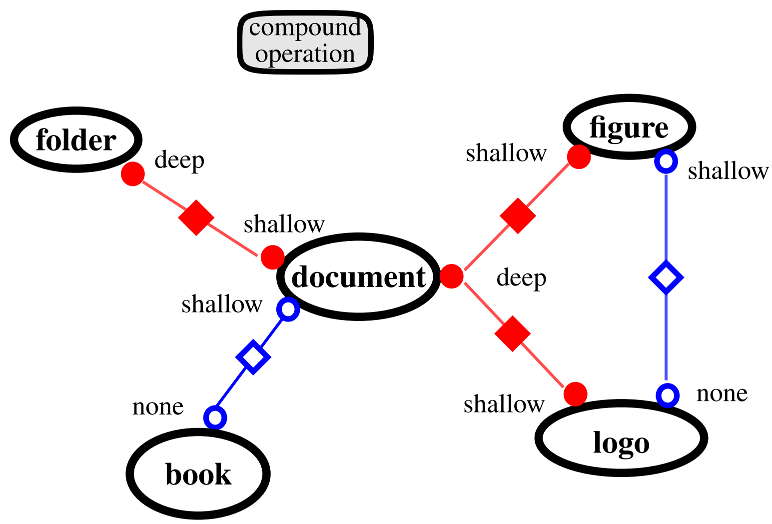


Figure 6-15 Prior to applying copy to the graph.

In this example, the copy is performed in two passes. The first pass creates a list representation of the relevant edges of the graph. The second pass takes the list as input, copies the relevant nodes and roles, then creates all the necessary links by copying the relevant relationships.

A compound copy request is initiated by issuing a *LifeCycleObjec*t::copy request on the folder. Since the folder participates in a graph of related objects, it creates an object supporting the *CosCompoundLifeCycle::Operations* interface (the *Operations* object). Then the folder issues a *CosCompoundLifeCycle::Operation*s::copy request on the *Operations* object, passing in its own *CosCompoundLifeCycle::Node* object reference as the starting node. The copy operation will copy the graph of related objects and return an object reference for the copy of the folder object.

The remainder of this section provides a description of how the *Operations* object might implement the copy operation.

First Pass of the Compound Copy Operation

The first pass consists of creating a list representation of the relevant edges of the graph. The *Operations* object uses an object supporting the *CosGraphs::Traversal* interface to do most of the work.

The *Operations* object creates an object supporting the *CosGraphs::TraversalCriteria* interface by calling *CosCompoundLifeCycle::PropagationCriteriaFactory*::create.

The *Operations* object then creates a *CosGraphs::Traversal* object by calling *CosGraphs::TraversalFactory*::create_traversal_on, passing in the object supporting the *CosGraphs::TraversalCriteria* interface. Calls on the *CosGraphs::Traversal* object yield an unordered list of *CosGraphs::Traversal*::*ScopedEdges* containing the following information.

(folder, ContainsRole, Containment, ContainedInRole, document)
(document, ReferencesRole, Reference, ReferencedByRole, book)
(document, ContainedInRole, Containment, ContainsRole, folder)
(document, ContainsRole, Containment, ContainedInRole, figure)
(document, ContainsRole, Containment, ContainedInRole, logo)
(figure, ReferencesRole, Reference, ReferencedByRole, logo)
(figure, ContainedInRole, Containment, ContainsRole, document)
(logo, ContainedInRole, Containment, ContainsRole, document)

This list will be referred to as the *OriginalEdgeList*.

Since the propagation value for copy from the document to the book is shallow, the traversal did not visit the book. As such, the edge:

(book, ReferencedByRole, Reference, References, document)

is not included. Although the traversal did visit the logo, the edge

(logo, ReferencedByRole, Reference, ReferencesRole, figure)

is not included because the propagation value for copy from the logo to the figure is none.

For more detailed information regarding the output of the *CosGraphs::Traversal* object with respect to the use of propagation semantics, see “Compound Operations” on page 9-36 of the Relationship Service.

Second Pass of the Compound Copy Operation

The second pass copies all the relevant nodes and then relates them by copying the relevant relationships.

First, the set of nodes to be copied must be determined. This consists of all the distinct nodes in the left column of the *OriginalEdgeList* . Since a node may be involved in multiple edges, it may appear multiple times in the list; it should only be copied once. Each node in this set is copied by issuing a *CosCompoundLifeCycle::Node*::copy_node request. This request will cause the node *and all of its roles* to be copied; the new node and its roles will be returned.

- For each returned role of the copied node, an entry is made in a table of new roles. Each entry consists of:
- The role object is the data and
- The node’s CosGraphs::Traversal::TraversalScopedId and the role’s CORBA::InterfaceDef together serve as a key.

The final step is to create all the relationships for the copied graph. All of the distinct relationships in the center column of the *OriginalEdgeList* need to be copied. Although a relationship may appear multiple times in the list, it should only be copied once. Each relationship is copied by issuing a *CosCompoundLifeCycle::Relationship*::copy_relationship request. The arguments to *CosCompoundLifeCycle::Relationship*::copy_relationship include the list of roles to be included in the new relationship. Some of these roles will be copies that were created as a result of processing deep propagation values; others will be roles in the original graph.

Thus, copy each unique relationship in the *OriginalEdgeList*, using NamedRoles as follows:

For each role in an entry in the *OriginalEdgeList*, make a role key using the node's TraversalScopedId and the role's CORBA::InterfaceDef to search the table of new roles.

a. If the role was copied, the key will find the role's copy. The role's *RoleName* is obtained from the entry in the *OriginalEdgeList*. The role's copy and the *RoleName* are combined to form a *CosGraphs::NamedRole* which will then be included in the list of *CosGraphs::NamedRole*s passed to the *CosCompoundLifeCycle::Relationship*::copy_relationship method.

b. If no copy is found, the original *CosGraphs::NamedRole* is used instead.

Once all the *Relationships* have been copied, the *CosCompoundLifeCycle::Operations*::copy method is done.

Figure 6-16 illustrates the result of applying copy to the graph, starting at the folder.

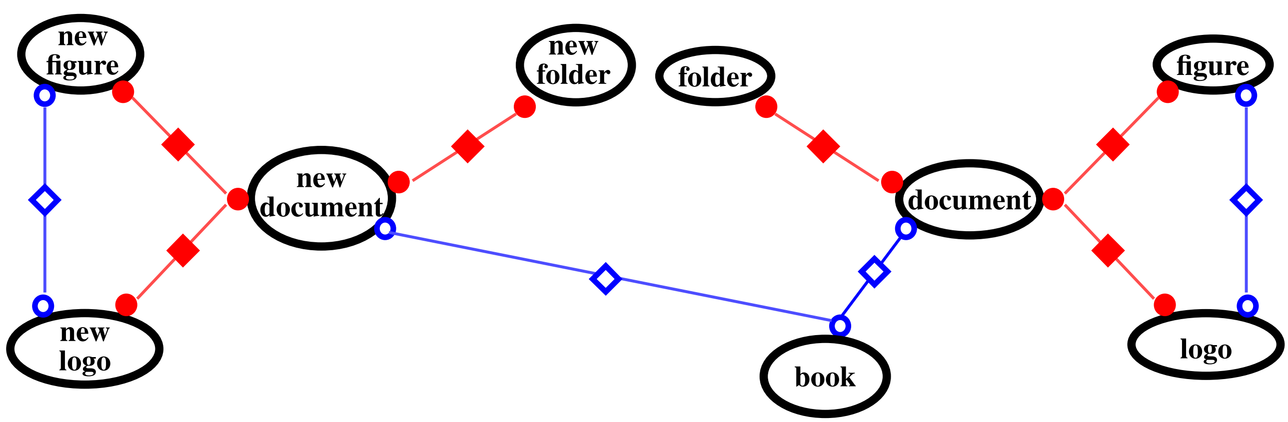


Figure 6-16 The result of applying copy to the graph, starting at the folder.

When the copy operation propagates to a node because of a deep propagation value, other shallow propagation values to that node are *promoted*. That is, they are processed as if they were deep; relationships are formed with the copied node, not with the original. This happened in the example; the shallow propagation value from the figure to the logo was promoted to deep because the logo was copied. As such, the new figure references the new logo, *not* the original logo.

A.4.2 The CosCompoundLifeCycle Module

The *CosCompoundLifeCycle* module defines

- The *Operations* interface for initiating compound life cycle operations on graphs of related objects,
- *OperationsFactory* interface for creating compound operations,
- The *Node*, *Role*, *Relationship* and *PropagationCriteriaFactory* interfaces for use by implementations of compound life cycle operations.

The *CosCompoundLifeCycle* module is given in Figure 6-17. Detailed descriptions of the interfaces follow.

```
#include <LifeCycle.idl>
#include <Relationships.idl>
#include <Graphs.idl>

module CosCompoundLifeCycle {
	interface OperationsFactory;
	interface Operations;
	interface Node;
	interface Role;
	interface Relationship;
	interface PropagationCriteriaFactory;

	enum Operation {copy, move, remove};

	struct RelationshipHandle {
		Relationship the_relationship;
		::CosObjectIdentity::ObjectIdentifier constant_random_id;
	};

	interface OperationsFactory {
		Operations create_compound_operations();
	};
```

Figure 6-17 The CosCompoundLifeCycle Module

```
interface Operations {
	Node copy (
			in Node starting_node,
			in ::CosLifeCycle::FactoryFinder there,
			in ::CosLifeCycle::Criteria the_criteria)
		raises (::CosLifeCycle::NoFactory,
			::CosLifeCycle::NotCopyable,
			::CosLifeCycle::InvalidCriteria,
			::CosLifeCycle::CannotMeetCriteria);
	void move (
			in Node starting_node,
			in ::CosLifeCycle::FactoryFinder there,
			in ::CosLifeCycle::Criteria the_criteria)
		raises (::CosLifeCycle::NoFactory,
			::CosLifeCycle::NotMovable,
			::CosLifeCycle::InvalidCriteria,
			::CosLifeCycle::CannotMeetCriteria);
	void remove (in Node starting_node)
		raises (::CosLifeCycle::NotRemovable);
	void destroy();
};

interface Node : ::CosGraphs::Node {
	exception NotLifeCycleObject {};
	void copy_node ( in ::CosLifeCycle::FactoryFinder there,
			in ::CosLifeCycle::Criteria the_criteria,
			out Node new_node,
			out Roles roles_of_new_node)
		raises (::CosLifeCycle::NoFactory,
			::CosLifeCycle::NotCopyable,
			::CosLifeCycle::InvalidCriteria,
			::CosLifeCycle::CannotMeetCriteria);
	void move_node (in ::CosLifeCycle::FactoryFinder there,
			in ::CosLifeCycle::Criteria the_criteria)
		raises (::CosLifeCycle::NoFactory,
			::CosLifeCycle::NotMovable,
			::CosLifeCycle::InvalidCriteria,
			::CosLifeCycle::CannotMeetCriteria);
	void remove_node ()
		raises (::CosLifeCycle::NotRemovable);
	::CosLifeCycle::LifeCycleObject get_life_cycle_object()
		raises (NotLifeCycleObject);
};
```

Figure 6-17 The CosCompoundLifeCycle Module *(Continued)*

```
    interface Role : ::CosGraphs::Role {
        Role copy_role (in ::CosLifeCycle::FactoryFinder there,
                in ::CosLifeCycle::Criteria the_criteria)
            raises (::CosLifeCycle::NoFactory,
                ::CosLifeCycle::NotCopyable,
                ::CosLifeCycle::InvalidCriteria,
                ::CosLifeCycle::CannotMeetCriteria);
        void move_role (in ::CosLifeCycle::FactoryFinder there,
                in ::CosLifeCycle::Criteria the_criteria)
            raises (::CosLifeCycle::NoFactory,
                ::CosLifeCycle::NotMovable,
                ::CosLifeCycle::InvalidCriteria,
                ::CosLifeCycle::CannotMeetCriteria);
        ::CosGraphs::PropagationValue life_cycle_propagation (
                in Operation op,
                in RelationshipHandle rel,
                in ::CosRelationships::RoleName to_role_name,
                out boolean same_for_all);
    };

    interface Relationship :
            ::CosRelationships::Relationship {
        Relationship copy_relationship (
                in ::CosLifeCycle::FactoryFinder there,
                in ::CosLifeCycle::Criteria the_criteria,
                in ::CosGraphs::NamedRoles new_roles)
            raises (::CosLifeCycle::NoFactory,
                ::CosLifeCycle::NotCopyable,
                ::CosLifeCycle::InvalidCriteria,
                ::CosLifeCycle::CannotMeetCriteria);
        void move_relationship (
                in ::CosLifeCycle::FactoryFinder there,
                in ::CosLifeCycle::Criteria the_criteria)
            raises (::CosLifeCycle::NoFactory,
                ::CosLifeCycle::NotMovable,
                ::CosLifeCycle::InvalidCriteria,
                ::CosLifeCycle::CannotMeetCriteria);
        ::CosGraphs::PropagationValue life_cycle_propagation (
                in Operation op,
                in ::CosRelationships::RoleName from_role_name,
                in ::CosRelationships::RoleName to_role_name,
                out boolean same_for_all);
    };

    interface PropagationCriteriaFactory {
        ::CosGraphs::TraversalCriteria create(in Operation op);
    };

};
```

Figure 6-17 The CosCompoundLifeCycle Module *(Continued)*

A.4.3 The OperationsFactory Interface

Creating a Compound Life Cycle Operation

```
Operations create_compound_operations();
```

The create_compound_operations operation creates an object that implements the compound life cycle operations, that is, the factory creates and returns an object that supports the *CosCompoundLifeCycyle::Operations* interface.

The Operations Interface

The *Operations* interface defines compound life cycle operations to copy, move and remove objects, given a starting node in a graph.

Applying the Copy Operation to a Graph of Related Objects

```
Node copy (
        in Node starting_node,
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotCopyable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
```

The copy operation applies the copy operation to a graph of related objects. The starting node is provided as the starting_node parameter. The copy should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If a node, role or relationship in the graph refuses to be copied, the NotCopyable exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

If appropriate factories to create a copies of the nodes and roles cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotCopyable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied graph, NO_RESOURCES will be raised.

It is implementation dependent whether this operation is *atomic*.

Applying the Move Operation to a Graph of Related Objects

```
void move (
        in Node starting_node,
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The move operation applies the move operation to a graph of related objects. The starting node is provided as the starting_node parameter. The migrated graph should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If a node, role or relationship in the graph refuses to be moved, the NotMovable exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

If appropriate factories to migrate the nodes and roles cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotMovable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated graph, NO_RESOURCES will be raised.

It is implementation-dependent whether this operation is *atomic*.

Applying the Remove Operation to a Graph of Related Objects

```
void remove (in Node starting_node)
    raises (::CosLifeCycle::NotRemovable);
```

The remove operation applies the remove operation to a graph of related objects. The starting node is provided as the starting_node parameter.

If a node, role or relationship in the graph refuses to be removed, the NotRemovable exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

It is implementation dependent whether this operation is *atomic*.

Destroying the Compound Operation

```
void destroy();
```

The destroy operation indicates to the compound operation that the client has completed operating on the graph. The compound operation object is destroyed.

The Node Interface

The *Node* interface defines operations to copy, move and remove a node.

Copying a Node

```
void copy_node ( in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria,
        out Node new_node,
        out Roles roles_of_new_node)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotCopyable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The copy operation makes a copy of the node and its roles. The new node and roles should be collocated with the factory finder given by the there parameter. The final input parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The result of a copy operation is a:

- *Node* object reference for the new node and
- Sequence of roles

Figure 6-18 illustrates the result of a copy. A node, when it is born, is not in any relationships with other objects. That is, the roles in the new node are "disconnected". It is the compound copy operation's job to correctly establish new relationships.

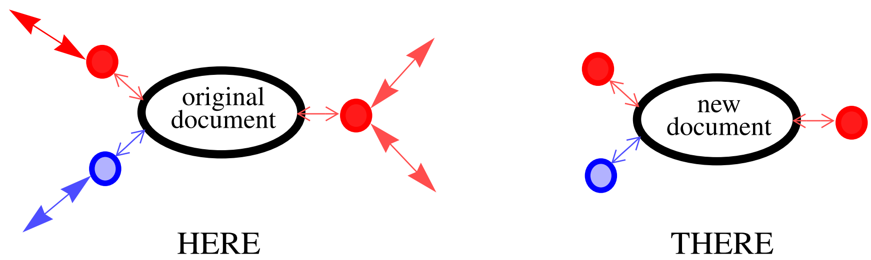


Figure 6-18 Copying a node returns the new object and the corresponding roles.

If the node or one of its roles refuses to be copied, the NotCopyable exception is raised with the node or role object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotCopyable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied node, NO_RESOURCES will be raised.

Moving a Node

```
void move_node (in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The move operation transfers some or all of the node's resources from "here" to "there". The move operation migrates a the node and its roles. The migrated node and roles should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the node or one of its roles refuses to be moved, the NotMovable exception is raised with the node or role object reference returned as a parameter to the exception.

If an appropriate factory to support migration "over there" cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotMovable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated node, NO_RESOURCES will be raised.

Removing a Node

```
void remove_node ()
    raises (::CosLifeCycle::NotRemovable);
```

The remove operation removes the node and its roles.

If the node or one of its roles refuses to be removed, the NotRemovable exception is raised with the node or role object reference returned as a parameter to the exception.

Getting the Node's Life Cycle Object

```
::CosLifeCycle::LifeCycleObject get_life_cycle_object()
    raises (NotLifeCycleObject);
```

Some nodes not only participate in the life cycle protocols for graphs of related objects but they also support the client's view of life cycle services. That is, the node also supports the *::CosLifeCycle::LifeCycleObject* interface described in the Life Cycle Service specification. The get_life_cycle_object operation returns the *::CosLifeCycle::LifeCycleObject* object reference for the node.

If the node does not support the *::CosLifeCycle::LifeCycleObject* interface, the NotLifeCycleObject exception is raised.

The Role Interface

The *Role* interface defines operations to copy and move a role. (The destroy operation is defined by the base Relationship Service. As such, there is no need to define a remove operation.) The *Role* interface also defines an operation to return the propagation values for the copy, move and remove operations.

The implementation of a *CompoundLifeCycle::Node* operation can call these operations on roles. For example, an implementation of copy on a node can call the copy operation on the *Role*.

Copying a Role

```
Role copy_role (in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria)
      raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotCopyable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
```

The copy operation makes a copy of the role. The new role should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The result of a copy operation is an object reference for the new object supporting the *Role* interface.

If the role refuses to be copied, the NotCopyable exception is raised with the role object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotCopyable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied role, NO_RESOURCES will be raised.

Moving a Role

```
void move_role (in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria)
      raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotMovable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
```

The move operation transfers some or all of the role’s resources. The move operation migrates the role. The migrated role should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the role refuses to be moved, the NotMovable exception is raised with the role object reference returned as a parameter to the exception.

If an appropriate factory to support migration cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotMovable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated role, NO_RESOURCES will be raised.

Getting a Propagation Value

```
::CosGraphs::PropagationValue life_cycle_propagation (
        in Operation op,
        in RelationshipHandle rel,
        in ::CosRelationships::RoleName to_role_name,
        out boolean same_for_all);
```

The life_cycle_propagation operation returns the propagation value to the role to_role_name for the life cycle operation op and the relationship rel. If the role can guarantee that the propagation value is the same for all relationships in which it participates, same_for_all is true.

The Relationship Interface

The *Relationship* interface defines operations to copy and move a relationship. (The destroy operation is defined by the Relationship Service. As such, there is no need to define a remove operation.) The *Relationship* interface also defines an operation to return the propagation values for the copy, move and remove operations.

Copying the Relationship

```
Relationship copy_relationship (
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria,
        in ::CosGraphs::NamedRoles new_roles)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotCopyable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The copy operation creates a new relationship. The new relationship should be collocated with the factory finder given by the there parameter. The second parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The values of the newly created relationship's attributes are defined by the implementation of this operation. However, the named_roles attribute of the newly created relationship must match new_roles. That is, the newly created relationship relates objects represented by new_roles parameter, not the by the original relationship's named roles.

The result of a copy operation is an object reference for the new object supporting the *Relationship* interface.

If the relationship refuses to be copied, the NotCopyable exception is raised with the relationship object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotCopyable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied role, NO_RESOURCES will be raised.

Moving the Relationship

```
void move_relationship (
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The move operation transfers some or all of the relationship's resources. The move operation migrates the relationship. The migrated relationship should be collocated with the factory finder given by the there parameter. The final parameter, the_criteria, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the relationship refuses to be moved, the NotMovable exception is raised with the relationship object reference returned as a parameter to the exception.

If an appropriate factory to support migration cannot be found, the NoFactory exception is raised. The exception value indicates the key used to find the factory.

In addition to the NoFactory and NotMovable exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated relationship, NO_RESOURCES will be raised.

Getting a Propagation Value

```
::CosGraphs::PropagationValue life_cycle_propagation (
        in Operation op,
        in ::CosRelationships::RoleName from_role_name,
        in ::CosRelationships::RoleName to_role_name,
        out boolean same_for_all);
```

The life_cycle_propagation operation returns the relationship's propagation value from the role from_role to the role to_role_name for the life cycle operation op. If the role named by from_role_name can guarantee that the propagation value is the same for all relationships in which it participates, same_for_all is true.

The PropagationCriteriaFactory Interface

The *CosGraphs* module in the Relationship Service defines a general service for traversing a graph of related objects. The service accepts a "call-back" object supporting the *::CosGraphs::TraversalCriteria* interface. Given a node, this object defines which edges to emit and which nodes to visit next.

The *PropgationCriteriaFactory* creates a *TraversalCriteria* object that determines which edges to emit and which nodes to visit based on propagation values for the compound life cycle operations.

Create a Traversal Criteria Based on Life Cycle Propagation Values

```
::CosGraphs::TraversalCriteria create(in Operation op);
```

The create operation returns a TraversalCriteria object for an operation op that determines which edges to emit and which nodes to visit based on propagation values for op. For a more detailed discussion see section A.4.1 of this appendix and "Traversing Graphs of Related Objects" on page 9-35 of the Relationship specification.

A.4.4 Specific Life Cycle Relationships

The Relationship service defines two important relationships, *containment* and *reference*. Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment is represented by a relationship with two roles: the *ContainsRole*, and the *ContainedInRole.* Similarly, reference is represented by a relationship with two roles: *ReferencesRole and ReferencedByRole*.

The compound life cycle specification adds life cycle semantics to these specific relationships. That is, it defines propagation values for containment and reference.

A.4.5 The CosLifeCycleContainment Module

The *CosLifeCycleContainment* module defines three interfaces
- the *Relationship* interface
- the *ContainsRole* interface and
- the *ContainedInRole* interface.

```
#include <Containment.idl>
#include <CompoundLifeCycle.idl>

module CosLifeCycleContainment {

        interface Relationship :
                ::CosCompoundLifeCycle::Relationship,
                ::CosContainment::Relationship {};

        interface ContainsRole :
                ::CosCompoundLifeCycle::Role,
                ::CosContainment::ContainsRole {};

        interface ContainedInRole :
                ::CosCompoundLifeCycle::Role,
                ::CosContainment::ContainedInRole {};
};
```

Figure 6-19 The CosLifeCycleContainment module

The *CosLifeCycleContainment* module does not define new operations. It merely “mixes in” interfaces from the *CosCompoundLifeCycle* and *CosContainment* modules. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
relationship_type	CosLifeCycleContainment::Relationship
degree	2
named_role_types	“ContainsRole”,CosLifeCycleContainment::ContainsRole; “ContainedInRole”,CosLifeCycleContainment::ContainedInRole

The CosRelationships::RelationshipFactory::create operation will raise DegreeError if the number of roles passed as arguments is not 2. It will raise RoleTypeError if the roles are not *CosLifeCycleContainment*::*ContainsRole* and

CosLifeCycleContainment::*ContainedInRole*. It will raise MaxCardinalityExceeded if the *CosLifeCycleContainment*::*ContainedInRole* is already participating in a relationship.

RoleFactory attribute for ContainsRole	**value**
role_type	CosLifeCycleContainment::ContainsRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosCompoundLifeCycle::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosLifeCycleContainment::Relationship* interface.

RoleFactory attribute for ContainedInRole	**value**
role_type	CosLifeCycleContainment::ContainedInRole
maximum_cardinality	1
minimum_cardinality	1
related_object_types	CosCompoundLifeCycle::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosCompoundLIfeCycle::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosLifeCycleContainment::Relationship* interface. The *CosRelationships::RoleFactory::*link operation will raise MaxCardinalityExceeded if it is already participating in a containment relationship.

The *CosLifeCycleContainment*::*ContainsRole*::life_cycle_propagation operation returns the following:

operation	**ContainsRole to ContainedInRole**
copy	deep
move	deep
remove	deep

The *CosLifeCycleContainment*::*ContainedInRole*::life_cycle_propagation operation returns the following::

operation	**ContainedInRole to ContainsRole**
copy	shallow
move	shallow
remove	shallow

A.4.6 The CosLifeCycleReference Module

The *CosLifeCycleReference* module defines three interfaces

- the *Relationship* interface,
- the *ReferencesRole* interface and
- the *ReferencedByRole* interface.

```
#include <Reference.idl>
#include <CompoundLifeCycle.idl>

module CosLifeCycleReference {

    interface Relationship :
        ::CosCompoundLifeCycle::Relationship,
        ::CosReference::Relationship {};

    interface ReferencesRole :
        ::CosCompoundLifeCycle::Role,
        ::CosReference::ReferencesRole {};

    interface ReferencedByRole :
        ::CosCompoundLifeCycle::Role,
        ::CosReference::ReferencedByRole {};
};
```

Figure 6-20 The CosLifeCycleReference module

The *CosLifeCycleReference* module does not define new operations. It merely “mixes in” interfaces from the *CosCompoundLifeCycle* and *CosReference* modules. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
relationship_type	CosLifeCycleReference::Relationship
degree	2
named_role_types	“ReferencesRole”,CosLifeCycleReference::ReferencesRole; “ReferencedByRole”,CosLifeCycleReference::ReferencedByRole

The CosRelationships::RelationshipFactory::create operation will raise DegreeError if the number of roles passed as arguments is not 2. It will raise RoleTypeError if the roles are not *CosReference::ReferencesRole* and *CosReference*::*ReferencedByRole*.

RoleFactory attribute for ReferencesRole	**value**
role_type	CosLifeCycleReference::ReferencesRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosCompoundLifeCycle::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosLifeCycleReference::Relationship* interface.

RoleFactory attribute for ReferencedByRole	**value**
role_type	CosLifeCycleReference::ReferencedByRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosCompoundLifeCycle::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosLifeCycleRelationship::Relationship* interface.

The *CosLifeCycleReference*::*ReferencesRole*::life_cycle_propagation operation returns the following:

operation	**ReferencesRole to ReferencedByRole**
copy	shallow
move	shallow
remove	shallow

The *CosLifeCycleReference*::*ReferencedByRole*::life_cycle_propagation operation returns the following::

operation	ReferencedByRole to ReferencesRole
copy	none
move	shallow
remove	shallow

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface.
The CosRelationships::RelationshipFactory::create operation will raise DegreeError if the number of roles passed as arguments is not 2. It will raise RoleTypeError if the roles are not *CosLifeCycleReference*::*ReferencesRole* and *CosLifeCycleReference*::*ReferencedByRole*.

A.5 References

1. James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations." *OOPSLA 1988 Proceedings*, pg. 285-296
2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, "Object-oriented Modeling and Design." Prentice Hall, 1991.

Appendix B Filters

Note – Appendix B is not part of the Life Cycle Services specification. It sketches a mechanism for expressing filters. This appendix is included to provided an example of how a filter might be provided.

A factory represents a scope of resource allocation, which is the set of resources available to the factory. Whenever it receives a creation request, a factory will allocate resources according to any policies which are in operation.

Clearly, by choosing a particular factory upon which to issue a create request, a client is exerting some control over the allocation of resources. Therefore, a client can limit the scope of resource allocation, by issuing the request on a different factory which represents a smaller set of resources.

However, there are two problems with this. Firstly, the granularity of resources may be much smaller than the granularity represented by the factories in a system. For example, there are unlikely to be factories which represent individual disk segments.

Secondly, the client may wish to rule out the use of particular resources within a scope, but avoid having a general reduction in scope. For example, the client might not be concerned with which machine within a LAN an object is created on, providing it is not on machine X.

Both of these needs can be addressed by providing a filter. In the first case, the filter is relatively simple; it will simply limit the scope of resource allocation. In the second case, the filter will need to be more sophisticated.

This appendix describes one way of providing filters using *properties* and *constraint expressions*. These concepts appear in the development of Trading in the ISO/IEC/CCITT Open Distributed Processing standards. Service providers register their service with the Trader and use properties to describe the service offer. Potential clients may then use a constraint expressions to describe the requirements which service offers must satisfy.

Similarly, the life cycle service may define a number of properties to represent the different kinds of resources available within in a system and clients may use constraint expressions to place the restrictions upon the use of those resources.

Note – The Object Services Architecture identifies an Object Properties Service which enables an object to have a set of arbitrary named values associated with it. These are very similar to the concept of properties as used in Trading and in this appendix.

B.1 Resources as Properties

Resource properties are application and generic factory implementation dependent and it is beyond the scope of this specification to identify standard properties which all generic factory implementations will recognize. The properties described in this appendix are given as examples only. Table 6-8 gives some examples of properties that might be supported by a generic factory.

Table 6-8 Examples of properties supported by a generic factory

Property Name	**Meaning**
Host	Host name of the machine
Architecture	Machine architecture, e.g. "intel", "sparc"
OSArchitecture	Operating system architecture e.g. "solaris", "hpux"

B.2 Constraint Expressions

Constraints are expressed in a Constraint Language which provides a set of operators which allow arbitrarily complex expressions involving properties and potential values to be specified. A property lists *satisfies* a constraint if the constraint expression is true when evaluated with respect to the property list.

Constraint expressions are very flexible. For example, if a client has an object executing on a machine called 'Host1' and wishes to create another object which is *not* on the same machine, the client can specify the constraint "Host != 'Host1'".

The constraint expression described here works with properties for which the value can be a string, a number, or a set of values.

The constraint language consists of:

- comparative functions: ==, !=, >, >=, <, <=, in
- constructors: and, or, not
- property names
- numeric and string constants
- mathematical operators: +, -, *, /
- grouping operators: (,), [,]

The following precedence relations hold in the absence of parentheses, in the order of lowest to highest:

- + and -
- * and /
- or
- and
- not

The comparative operator in checks for the inclusion of a particular string constant in the list which is the value of a property.

B.3 BNF for Constraint Expressions

```
<ConstraintExpr>    :=    [ <Expr> ]

<Expr>              :=    <Expr> ”or” <Expr>
                    |     <Expr> ”and” <Expr>
                    |     ”not” <Expr>
                    |     ”(” <Expr> ”)”
                    |     <SetExpr> <SetOp> <SetExpr>
                    |     <StrExpr> <StrOp> <StrExpr>
                    |     <NumExpr> <NumOp> <NumExpr>
                    |     <NumExpr> ”in” <SetExpr>
                    |     <StrExpr> ”in” <SetExpr>

<NumOp>             :=    ”==” | ”!=” | ”<” | ”<=” | ”>” | ”>=”

<StrOp>             :=    ”==” | ”!=”

<SetOp>             :=    ”==” | ”!=”


<NumExpr>           :=    <NumTerm>
                    |     <NumExpr> ”+” <NumTerm>
                    |     <NumExpr> ”-” <NumTerm>

<NumTerm>           :=    <NumFactor>
                    |     <NumTerm> ”*” <NumFactor>
                    |     <NumTerm> ”/” <NumFactor>

<NumFactor>         :=    <Identifier>
                    |     <Number>
                    |     ”(” <NumExpr> ”)”
                    |     ”-” <NumFactor>

<StrExpr>           :=    <StrTerm>
                    |     <StrExpr> ”+” <StrTerm>

<StrTerm>           :=    <Identifier>
                    |     <String>
                    |     ”(” <StrExpr> ”)”

<SetExpr>           :=    <SetTerm>
                    |     <SetExpr> ”+” <SetTerm>

<SetTerm>           :=    <Identifier>
                    |     <Set>
                    |     ”(” <SetExpr> ”)”

<Identifier>        :=    <Word>
```

```
<Number>      :=   <Integer>
              |    <Float>

<Integer>     :=   { <Digit> }+

<Float>       :=   <Mantissa> [ <Sign> ] [ <Exponent> ]

<Mantissa>    :=   <Integer> [ ”.” [ <Integer> ] ]
              |    ”.” <Integer>

<Sign>        :=   ”-”
              |    ”+”

<Exponent>    :=   ”e” <Integer>
              |    ”E” <Integer>

<Word>        :=   <Letter> { <AlphaNum> }*

<AlphaNum>    :=   <Letter>
              |    <Digit>
              |    ”_”

<String>      :=   ”’” { <Char> }* ”’”

<Char>        :=   <Letter>
              |    <Digit>
              |    <Other>

<Set>         :=   ”{” <Elements> ”}”

<Elements>    :=   [ <Element> { <Sp>+ <Element> }* ]

<Element>     :=   <Number>
              |    <Word>
              |    <String>

<Letter>      :=   a | b | c | d | e | f | g | h | i | j | k
              |    l | m | n | o | p | q | r | s | t | u | v
              |    w | x | y | z | A | B | C | D | E | F | G
              |    H | I | J | K | L | M | N | O | P | Q | R
              |    S | T | U | V | W | X | Y | Z

<Digit>       :=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Other>       :=   <Sp> | ~ | ! | @ | # | $ | % | ^ | & | * | (
              |    ) | - | _ | = | + | [ | { | ] | } | ; | :
              |    “ | \ | | | , | < | . | > | / | ?

<Sp>          :=   ” ”
```

Appendix C Administration

Note – Appendix C is not part of the Life Cycle Services specification. This description is included as a suggested way of administering generic factories.

The specification for the life cycle service includes the *GenericFactory* interface. There will be at least two styles of object which support that interface:

- implementation specific factories that actually assemble the resources for a new object, and
- generic factories which pass requests on to either implementation specific factories or other generic factories.

By configuring generic factories and implementation specific factories into a graph, a creation service can be built which administers the allocation of a large number of resources and can use them to create a wide variety of objects.

To ensure that the creation service is scalable, it is essential that the principle of *federation* is adopted – each component retains its autonomy rather than becoming subordinate to another.

Whenever the creation service receives a creation request, the request will need to traverse the graph until it reaches an implementation specific factory which can satisfy the request. As the request traverses the graph, each non-terminal node in the graph (i.e. the generic factories) will decide which link the request will traverse next. Decisions will be based upon information about each available link, any *policies* in force at that node and, of course, the actual request.

Clearly, the configuration and policies of such a creation service will need to be administered. However, the specification does not include the specification of an administration interface. This is because the principle of federation is not only important to the life cycle service. It will be essential to a number of other services, notably trading, and the OMG plans to address the issue of federation for *all* object services, rather than making a premature specification addressing the needs of just one service.

The remainder of this appendix describes the principle of federation in more detail, outlines the use of policies and preferences to support federation, and then concludes with a suggestion for how an administration interface might look.

C.1 Federation

Federation is essential in large-scale distributed systems where the existence of centralized ownership and universal control cannot be assumed. In these systems the only way to achieve cooperation between autonomous systems without creating a hierarchical structure is to use federation. Federation is also beneficial to smaller systems which can exploit the high degree of flexibility which federation provides.

Federation differs from the more conventional approach of adopting a strictly hierarchical organization in a number of ways. Firstly, components can provide their service to any number of others, not just the single component which is its “parent” in the hierarchy. Secondly, components can establish peer-to-peer relationships, eliminating the need for a single component at the top of the hierarchy. Finally, this approach avoids the necessity of maintaining a global namespace. Instead, all names are relative to the context in which they are used.

Federation enables previously distinct systems to be unified without requiring global changes to their naming structures and system management hierarchies. The administration functions must ensure the systems are configured appropriately, e.g. avoiding circular references in those graphs which must be kept acyclic.

C.1.1 Federation in Object Services

In addition to the use of federation in configuring generic factories, federation is also applicable to a number of other services.

Trading is a notable example. A global offer space is neither practical nor desirable. Consequently, there will be multiple traders, each representing a different portion of the offer space. Offers held by one trader can be made available to the clients of another trader through federation.

The naming service specification also demonstrates attributes of federation. Naming contexts can be bound to other naming contexts and requests for name resolution can be passed across the links. However, it is entirely the concern of the naming context how it resolves the name within its domain, i.e. it is autonomous.

C.1.2 Federation Issues

There are a number of issues which need to be addressed for federation to be used in a cohesive fashion across all object services.

Visibility of the Federation Graph

The naming service makes the configuration of naming contexts into a graph very visible to the clients. This is essential, because the naming service must provide clients with a structured namespace.

On the other hand, it is not clear that a client should ever be able to see the internal structure of a life cycle creation service built with generic and implementation specific factories.

The trading service falls in between the two extremes. It may be useful for a client to be able to navigate the structure of a trading service graph in order to have more control over the visibility of offers. However, this may make clients too dependent upon the organization of the trading service and limit the flexibility of the system administrator in reorganizing the trading service to provide the most effective service.

Service Interface vs. Administration Interface

In general, it is desirable to federate using the service interface for the links and reserve the administration interface for the administrators. This approach ensures that autonomy is retained. However, this precludes the use of compound names in the administration functions because the administration functions cannot traverse the graph; only simple names can be used in administration only functions.

However, this is inappropriate for services where graph manipulation is an essential part of the service. For example, the naming service specification does not distinguish between administration functions for manipulating the graph and service functions. This is clearly correct; the clients need to be able to manipulate the graph by creating, binding and destroying contexts.

Multiple Service Interfaces

A node in a federation graph may be a conspiracy and offer multiple service interfaces, perhaps one for each point it is bound into the graph. However, for services where the administration is kept distinct from the service, it is likely that the conspiracy will support only one administration interface.

In these situations, it becomes necessary for an administrator to be able to match service interfaces to conspiracies, i.e. to match one or more service interfaces to an administrative interface. The example in Section C.3 provides a solution to this which, in theory, will scale, but there may be better ways of doing this.

Cycles and Peer-to-Peer Relationships

The introduction of cycles into a federation graph is a contentious issue. Since peer-to-peer relationships are a degenerate form of cycle, any service which supports peer-to-peer relationships must be capable of handling cycles. The major impact of this is to provide loop detection on operations which would otherwise go out of control. Both trading and naming services are examples of this kind of service.

However, some services may not be able to handle cycles effectively and will wish to proscibe them. This probably covers peer-to-peer relationships, although that might be an acceptable special case. An example of this might be the life cycle creation service, where information about the current usage of the available resources must percolate up the graph in order to make informed decisions, but the introduction of cycles would make this information unclear or even meaningless.

C.2 Policies

It is frequently necessary to configure the way in which operations are performed in order to tune the performance, e.g how long a search operation may take, how many matches can be returned, or how much memory to use for a cache.

The same problems exist in distributed systems except that such configuration parameters must be explicitly passed around. Where different administrative domains are connected, such configuration parameters cannot be enforced by one domain on the other. Similarly, users may want to control the configuration but must be prevented from hogging resources, e.g memory, disk space, etc. Some configuration elements must be enforced, e.g disk quotas, some elements may specify defaults which can be changed and some elements may be requests which may or may not clash with hard limits e.g max memory per process.

Policies are used as a generic solution to this problem – wherever some kind of choice needs to be made, policies may be used to guide the decision making process.

Table 6-9 provides some examples of policies. which a federated service might support.

Table 6-9 Example policies

Policy Name	Meaning
search_algorithm	determines whether the federation graph should be traversed in a depth first or breadth first fashion.
cross_ boundaries	determines whether administrative boundaries should be crossed.
maximum_distance	how far to traverse a graph before failing a request.

When invoking operations, clients can specify preferences for particular policies. Providing the service has no value set for that policy, the preference will be simply added to the policy list for the duration of the request. However, if a service policy is already specified then the preference will either be ignored or, for policies such as “maximum_distance”, the more constraining value will be adopted.

As a request traverses a graph, each node will pass its current policy set as preferences. In this way, the autonomy of individual administrative domains is preserved.

When an object doesn’t implement all choices of a policy, it should not allow its policy to be modified to an unsupported value. This means that implementation limitations are handled as Administrative hard limits which provides the correct semantics.

Where no policy is specified by either administrator or client, the implementation determines its own behavior. However, this decision would not be propagated through the graph (as a preference), leaving it to each node in the graph to make its own decision.

C.3 An Example LifeCycleService Module

Administrators access the administration functions via the *LifeCycleService* module, which defines the *LifeCycleServiceAdmin* interface. This example is intended to work with the *GenericFactory* interface in the specification. As a result, the administration functions cannot make use of compound names.

```
#include “LifeCycle.idl”

module LifeCycleService {

    typedef sequence <Lifecycle::NameValuePair> PolicyList;
    typedef sequence <Lifecycle::Key> Keys;
    typedef sequence <Lifecycle::NameValuePair> PropertyList;
    typedef sequence <Naming::NameComponent> NameComponents;

    interface LifeCycleServiceAdmin {

        attribute PolicyList policies;

        void bind_generic_factory(
                in Lifecycle::GenericFactory gf,
                in Naming::NameComponent name,
                in Keys key_set,
                in PropertyList other_properties)
            raises (Naming::AlreadBound, Naming::InvalidName);

        void unbind_generic_factory(
                in Naming::NameComponent name)
            raises (Naming::NotFound, Naming::InvalidName);

        Lifecycle::GenericFactory resolve_generic_factory(
                in Naming::NameComponent name)
            raises (Naming::NotFound, Naming::InvalidName);

        NameComponents list_generic_factories();

        boolean match_service (in Lifecycle::GenericFactory f);

        string get_hint();

        void get_link_properties(
                in Naming::NameComponent name,
                out Keys key_set,
                out PropertyList other_properties)
            raises (Naming::NotFound, Naming::InvalidName);
    };

};
```

Figure 6-20 The LifeCycleService Module

C.3.1 The LifeCycleServiceAdmin Interface

The *LifeCycleServiceAdmin* interface provides the basic administration operations required to enable the lifecycle service to be administered by a set of tools or an administration service. The operations enable configuration of factories supporting the *GenericFactory* interface into a graph and setting of policies for those factories.

bind_generic_factory

```
void bind_generic_factory(
        in Lifecycle::GenericFactory gf,
        in Naming::NameComponent name,
        in Keys key_set,
        in PropertyList other_properties)
    raises (Naming::AlreadBound, Naming::InvalidName);
```

This operation binds a factory supporting the *GenericFactory* interface into a graph. The name must be unique within the context of the target of the operation. From then on, that factory can be identified by that name.

In order to make a good decision about which link to choose for a request, the node needs to be provided with additional information about those factories. This information may be fairly dynamic, e.g. the current usage of the resources available through the link, or more static, e.g. the Keys for which the link can provide support.

The key_set parameter is a list of the keys for which the factory can provide support. In the case of an implementation specific factory, this list will often only have one member.

The other_properties parameter can be used to provide other static properties associated with the factory. For example, an “Architectures” property would indicate the type(s) of machine which the factory could create objects on.

Changes to the static information as well as more dynamic information can be monitored through the Events service. Each factory would generate events whenever the information changed significantly (e.g. a new *GenericFactory* interface with new keys is bound to the factory, or there is a change in the usage of resources available to the factory) and these can then be passed to those factories which need to know.

unbind_generic_factory

```
void unbind_generic_factory(
        in Naming::NameComponent name)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation unbinds the generic factory identified by the name.

resolve_generic_factory

```
Lifecycle::GenericFactory resolve_generic_factory(
        in Naming::NameComponent name)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation takes the name supplied and returns the reference to the *GenericFactory* object.

list_generic_factories

```
NameComponents list_generic_factories();
```

This operation returns a list of the names of all the bound factories.

match_service

```
boolean match_service (in Lifecycle::GenericFactory f);
```

This operation returns true if the generic factory interface is supported by the target.

get_hint

```
string get_hint();
```

This operation returns a hint associated with the target, see *Building a Map of a Graph* below.

get_link_properties

```
void get_link_properties(
        in Naming::NameComponent name,
        out Keys key_set,
        out PropertyList other_properties)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation returns the key_set and other_properties associated with the name.

Building a Map of a Graph

Administration tools may wish to build a map of a federation graph from scratch and some of the operations above are provided for that purpose.

First of all, the tool must obtain the set of administration interfaces for all the factories to be administered. These might be obtained from a number of sources, e.g. a well-known trading context.

For each interface, the list_generic_factories operation obtains a list of all the links for each node. Using resolve_generic_factory, a service interface can be obtained for each link. These can then be matched to an administration interface using match_service.

Clearly, this does not scale well if there are many nodes involved because of the average number of invocations of match_service required. This problem can be solved if one of the other_properties associated with each service interface is a *hint* and a hint is available for each administration interface. If the hints are the same, there may be a match and match_service is called to check. If the hints could be guaranteed to be unambiguous, the invocation could be avoided altogether, but this requires a global namespace for the hints. The best that can reasonably be achieved is to reduce the chance of a clash to a minimum.

The get_hint and get_link_properties can be used for this purpose.

Appendix D Support for PCTE Objects[5]

Note – Appendix D is not part of the Life Cycle Services specification. This appendix defines a set of criteria[6] suitable for supporting PCTE objects.

It is intended that objects in a PCTE repository be among those objects that can be managed though this lifecycle interface. It is reasonable to expect that applications written for PCTE will use the PCTE APIs to manage the life-cycle of PCTE objects. It is also reasonable to expect that clients not specifically written for relationship-oriented objects will not be able to manipulate the life-cycles of PCTE objects. However, between these two, one can envision clients which desire to be flexible, working on objects which may or may not be stored in the PCTE repository. One can also envision object factories, constructed to make use of PCTE which provide services to clients that are not PCTE applications because they do not have the appropriate working schemas, etc.

Support for these clients employs a series of conventional interpretations of the lifecycle operations. This appendix provides one such set of conventions to demonstrate the feasibility of the use of these interfaces in a context supporting PCTE.

Object references appear in constraint expressions in the form of character strings. Any implementation of PCTE as a CORBA Object Adapter has to establish a relationship between these and the corresponding CORBA types, and be able to convert between them.

D.1 Overview

A PCTE repository can be viewed as a generic factory. Using whatever naming or trading services are appropriate, a client wishing to use the PCTE factory obtains an object reference to it. To support the simple applications intending to operate within the context of a single PCTE repository, the PCTE factory supports the operations defined by both the *GenericFactory* and *FactoryFinder* interfaces. The client can then invoke the PCTE factory's create_object operation, or pass the factory as the "factory finder" when invoking the move or copy operations to move or copy within the same PCTE repository. These clients include the servers implementing the move and copy operations for various PCTE objects as well.

5. PCTE details used here are from the PCTE Abstract Specification, Standard ECMA-149 available from the European Computer Manufacturers Association.

6. As defined in section 6.2.4 of the life cycle specification.

Lifecycle creation, copy, and move operations are influenced by a sequence of criteria. Criteria are specified as a sequence of name/value pairs. Certain criteria are of interest to the PCTE factories:

"logical location"

The logical location is used to express the logical connection information that must be specified when creating or copying a PCTE object. Logical location is a sequence of name/value pairs expressing a connection for the object. The PCTE factory supports and requires two:

ORIGIN	A string representation of the reference to the object to which the newly created object is to be connected.
ORIGINLINK	The name of the origin object's link which is to hold the link from the origin object to the newly created object.

"filter"

The filter is used to express the fact that an object being created, copied, or moved should reside on the same volume as some other, nearby, object. A filter is an expression as described in B.3. For PCTE, the term "NEAR=" followed by an object reference to the designated nearby object indicates that the new object is to be located at least as near as the same volume to the specified object. "authorization" Although omitted from table 1-4 because no proposal on authorization has yet been accepted by OMG, this lifecycle criterion is required to create PCTE objects.

D.2 *Object Creation*

The LifeCycle::GenericFactory::create_object operation in this specification is borne by factory objects. It has two parameters:

1. a key used to identify the desired object to be created and
2. a set of criteria expressed in an NVP-list.

The corresponding PCTE operation is called OBJECT_CREATE. The parameters to OBJECT_CREATE are obtained from the LifeCycle::GenericFactory::create_object parameters.

The PCTE operation OBJECT_CREATE has six parameters:

1. the type of object to be created This is the "key" from LifeCycle create_object.
2. the origin object of the relation anchoring the new object This is the object identified as the named "ORIGIN" of the logical location criterion.
3. the name of the link from that origin object to the new object This is the string identified as the named "ORIGINLINK" of the logical location criterion.
4. an optional key for that link This is the string identified as the named "LINKKEY" of the initialization criteria.

5. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.

6. an access mask This is the string identified as the named “ACCESS” of the authorization criteria This string is a simple mapping of the granted and denied access rights.

Exceptions raised by PCTE are mapped to suitable LifeCycle exceptions.

D.3 Object Deletion

The LifeCycle::LifeCycleObject::remove operation in this specification is borne by all life-cycle objects. It has no parameters.

The corresponding PCTE operation is called OBJECT_DELETE. The parameters to OBJECT_DELETE are obtained from the object to be deleted using information about that object defined in PCTE’s schema information about the object.

The PCTE operation OBJECT_DELETE has two parameters:

1. the origin object of a relation anchoring the object to be deleted and

2. the name of the link from that origin object to the object to be deleted.

To both ensure that the controlling object is actually deleted and maintain the PCTE referential integrity constraints the following steps are performed for each reversible link emanating from the controlling object:

1. Determine the object, o, that the link refers to.

2. Determine the name, r&prime., of the reverse link back from o.

3. Perform PCTE OBJECT_DELETE(o, r&prime.)

The objective is accomplished when all outgoing, reversible links have been dealt with thus, or before that if one of the OBJECT_DELETE calls fails because the object has already been deleted.

Exceptions raised by PCTE are mapped to suitable LifeCycle exceptions.

D.4 Object Copying

The LifeCycle::LifeCycleObject::copy operation in this specification is borne by all life-cycle objects. It has two parameters:

1. a factory-finder to assist in locating a factory that provides resources for the copied object

2. a set of criteria expressed in an NVP-list

The corresponding PCTE operation is called OBJECT_COPY. Some of the parameters to OBJECT_COPY can be obtained directly from the LifeCycle copy parameters. Other required information is obtained from the constraint expression parameter of the LifeCycle copy.

The PCTE operation OBJECT_COPY has six parameters:

1. the object to be copied This is the bearer object of LifeCycle copy operation.
2. the origin object of the relation anchoring the new object This is the object identified as the named “ORIGIN” of the logical location criterion.
3. the name of the link from that origin object to the new object This is the string identified as the named “ORIGINLINK” of the logical location criterion.
4. an optional key for that link This is the string identified as the named “LINKKEY” of the initialization criteria.
5. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.
6. an access mask This is the string identified as the named “ACCESS” of the authorization criteria This string is a simple mapping of the granted and denied access rights.

The semantics of the copy operation corresponds to the PCTE OBJECT_COPY semantics. They are based upon details of the object types involved, including which attributes, links and destination objects are “duplicable”.

Exceptions raised by PCTE are mapped to suitable CORBA standard exceptions.

D.5 Object Moving

The LifeCycle::LifeCycleObject::move operation in this specification is borne by all life-cycle objects. It has two parameters:

1. a factory-finder to assist in locating a factory that provide resources for the moved object
2. a set of criteria expressed in an NVP-list

The corresponding PCTE operation is called OBJECT_MOVE. The parameters to OBJECT_MOVE can be obtained directly from the LifeCycle copy parameters or from defaults.

The PCTE operation OBJECT_MOVE has three parameters:

1. the object to be copied This is the bearer object of LifeCycle move operation.
2. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.
3. scope - whether to move the object itself or the object and all its components

This will be defaulted to ATOMIC.

Concurrency Control Service 7

7.1 Service Description

The purpose of the Concurrency Control Service is to mediate concurrent access to an object such that the consistency of the object is not compromised when accessed by concurrently executing computations.

The Concurrency Control Service consists of multiple interfaces that support both transactional and non-transactional modes of operation. The user of the Concurrency Control Service can choose to acquire locks in one of two ways:

- On behalf of a transaction (transactional mode.) The Transaction Service drives the release of locks as the transaction commits or aborts.
- By acquiring locks on behalf of the current thread (that must be executing outside the scope of a transaction). In this non-transactional mode, the responsibility for dropping locks at the appropriate time lies with the user of the Concurrency Control Service.

The Concurrency Control Service ensures that transactional and non-transactional clients are serialized. Hence a non-transactional client that attempts to acquire a lock (in a conflicting mode) on an object that is locked by a transactional client will block until the transactional client drops the lock.

7.1.1 Basic Concepts of Concurrency Control

Clients and Resources

The Concurrency Control Service enables multiple *clients* to coordinate their access to shared *resources*. Coordinating access to a resource means that when multiple, concurrent clients access a single resource, any conflicting actions by the clients are reconciled so that the resource remains in a consistent state.

The Concurrency Control Service does not define what a resource is. It is up to the clients of the Concurrency Control Service to define resources and to properly identify potentially conflicting uses of those resources. In a typical use, an object would be a resource, and the object implementation would use the concurrency control service to coordinate concurrent access to the object by multiple clients.

Transactions as Clients

The Concurrency Control Service differentiates between two types of client: a transactional client and a non-transactional client. Conflicting access by clients of different types is managed by the Concurrency Control Service, thereby ensuring that clients always see the resource in a consistent state.

The Concurrency Control Service does not define what a transaction is. Transactions are defined by the Transaction Service. The Concurrency Control Service is designed to be used with the Transaction Service to coordinate the activities of concurrent transactions.

The Transaction Service supports two modes of operation: implicit and explicit. When operating in the implicit mode, a transaction is implicitly associated with the current thread of control. When executing in the explicit mode, a transaction is specified explicitly by the reference to the coordinator that manages the current transaction. To simplify the model of locking supported by the Concurrency Control Service when a transactional client is operating in the implicit transaction mode, transactional clients are limited to a single thread per transaction (nested transactions can be used when parallelism is necessary) and that thread can be executing on behalf of at most one transaction at a time.

Locks

The Concurrency Control service coordinates concurrent use of a resource using locks. A lock represents the ability of a specific client to access a specific resource in a particular way. Each lock is associated with a single resource and a single client. Coordination is achieved by preventing multiple clients from simultaneously possessing locks for the same resource if the activities of those clients might conflict. To achieve coordination, a client must obtain an appropriate lock before accessing a shared resource.

Lock Modes

The Concurrency Control Service defines several *lock modes*, which correspond to different categories of access. Having a variety of lock modes allows more flexible conflict resolution. For example, providing different modes for reading and writing allows a resource to support multiple concurrent clients that are only reading the data of the resource. The Concurrency Control Service also defines *intention locks* that support locking at multiple levels of granularity.

Lock Granularity

The Concurrency Control Service does not define the granularity of the resources that are locked. It defines a *lock set*, which is a collection of locks associated with a single resource. It is up to clients of the Concurrency Control Service to associate a lock set with each resource. Typically, if an object is a resource, the object would internally create and retain a lock set. However, the mapping between objects and resources (and lock sets) is up to the object implementation; the mapping could be one to one, but it could also be one to many, many to many, or many to one.

Conflict Resolution

A client obtains a lock on a resource using the Concurrency Control Service. The service will grant a lock to a client only if no other client holds a lock on the resource that would conflict with the intended access to the resource. The decision to grant a lock depends upon the modes of the locks held or requested. For example, a read lock conflicts with a write lock. If a write lock is held on a resource by one client, a read lock will not be granted to another client.

Conflict Resolution for Transactions

The decision to grant a lock also depends upon the relationships among the transactions that hold or request a lock. In particular, if the transactions are related by nesting (nested transactions), a lock may be granted that would otherwise be denied.

Lock Duration

Typically, a transaction will retain all of its locks until the transaction is completed (either committed or aborted). This policy supports serializability of transactional operations. Using the two phase commit protocol, locks held by a transaction are automatically dropped when the transaction completes.

There are also situations where levels of isolation that are weaker than serializability are acceptable, such as when an application does not want other applications to change an object while reading it and does not refer to the object again within the transaction. In these circumstances, it is acceptable to release locks before the containing transaction completes, hence the duration will be shorter than the containing transaction.

To manage the release of the locks held by a transaction, the Concurrency Control service defines a lock coordinator. Lock sets that are related (for example, by being created by a resource manager for resources of the same type) and that should drop their locks together when a transaction commits or aborts may share a lock coordinator. It is up to clients of the concurrency control service to associate lock sets together and to release the locks when a transaction commits or aborts.

7.2 Locking Model

This section covers a number of important issues that relate to the locking model supported by the Concurrency Control Service. For a complete discussion of these issues the reader is directed to one of the standard texts on the subject[1].

The Lock Modes section applies to clients that operate in both transactional and non-transaction modes. The Multiple Possession Semantics, Two-Phase Transactional Locking, and Nested Transaction sections are relevant only to clients that operate in transactional mode.

7.2.1 Lock Modes

Read, Write, and Upgrade Locks

The Concurrency Control service defines *read* (R) and *write* (W) lock modes that support the conventional multiple readers, one writer policy. Read locks conflict with write locks, and write locks conflict with other write locks.

In addition, the Concurrency Control service defines an *upgrade* (U) mode. An upgrade mode lock is a read lock that conflicts with itself. It is useful for avoiding a common form of deadlock that occurs when two or more clients attempt to read and then update the same resource. If more than one client holds a read lock on the resource, a deadlock will occur as soon as one of the clients requests a write lock on the resource. If each client requests a single upgrade lock followed by a write lock, this deadlock will not occur.

Intention Read and Intention Write Locks

The granularity of the resources locked by an application determines the concurrency within the application. Coarse granularity locks incur low overhead (since there are fewer locks to manage) but reduce concurrency since conflicts are more likely to occur. Fine granularity locks improve concurrency but result in a higher locking overhead since more locks are requested. Selecting a suitable lock granularity is a balance between the lock overhead and the degree of concurrency required. Using the Concurrency Control service, an application can be developed to use coarse or fine granularity locks by defining the associated resources appropriately.

In addition, the Concurrency Control service supports variable granularity locking using two additional lock modes, *intention read* (IR) and *intention write* (IW). These additional lock modes are used to exploit the natural hierarchical relationship between locks of different granularity.

For example, consider the hierarchical relationship inherent in a database: a database consists of a collection of files, with each file holding multiple records. To access a record, a coarse grain lock may be set on the database, but at the cost of restricting other clients from accessing the database. Clearly, this level of locking is unsuitable. However, only setting a lock on the record is also inappropriate, because another client might set a lock on the file holding the record and delete or modify the file.

Using variable granularity locking, a client first obtains intention locks on the ancestor(s) of the required resource. To read a record in the database, for example, the client obtains an intention read lock (IR) on the database and the file (in this order) before obtaining the read lock (R) on the record. Intention read locks (IR) conflict with write locks (W), and intention write locks (IW) conflict with read (R) and write (W) locks.

---

1. See *Concurrency Control and Recovery in Database Systems* by P.A. Bernstein, V. Hadzilacos, and N. Goodman, or *Transaction Processing: Concepts and Techniques* by J.N. Gray and A. Reuter.

Lock Mode Compatibility

Table 1, "Lock Compatibility," on page 5 defines the compatibility between the various

Table 1: Lock Compatibility

Granted Mode	Requested Mode				
	IR	R	U	IW	W
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

locking modes (the symbol * is used to indicate when locks conflict). When a client requests a lock on a resource that cannot be granted because another client holds a lock on the resource in a conflicting mode, the client must wait until the holding client releases its lock. The Concurrency Control Service enforces a queueing policy such that all clients waiting for a new lock are serviced in a first in, first out order, and subsequent requests are blocked by the first request waiting to be granted the lock, unless the requesting client is a transaction that is a member of the same transaction family as an existing holder of the lock.

7.2.2 Multiple Possession Semantics

The Concurrency Control Service interface supports a locking model called multiple possession semantics. In this model, a client can hold multiple locks on the same resource simultaneously. The locks can be of different modes. In addition, a client can hold multiple locks of the same mode on the same resource; effectively, a count is kept of the number of locks of a given mode that have been granted to the client. When a client holds locks on a resource in more than one mode, other clients will not be granted a lock on the resource unless the requested lock mode is compatible with all of the modes of the existing locks.

In contrast, using the conventional locking model,[2] when a client holding a lock on a resource requests a lock on the same resource in a stronger mode, the existing lock is promoted from the weaker mode to the stronger mode (once the stronger lock can be granted without causing a conflict). Since lock modes form only a partial order, there will not always be a stronger mode; in cases where neither mode is stronger, the lock will be promoted to the weakest mode that is at least as strong as either of the two modes.

7.3 Two-Phase Transactional Locking

The Concurrency Control Service provides primitives to support transaction-duration locking. Transaction duration locking is a special case of strict two-phase locking. In the first phase (the growing phase), a transaction obtains locks that are kept until the second phase (the shrinking phase), at which point they are released. A transaction must not release locks during the first phase, and must not obtain new locks during the second phase, otherwise concurrent computations may be able to view intermediate results of the transaction.

Two-phase locking is sufficient to guarantee serializability, hence this technique is used by transactions. During the normal execution of a transaction, no locks will be automatically dropped before the end of the transaction. When the transaction completes, the Concurrency Control Service must be informed so that the locks the transaction holds may be released. While releasing locks, no new locks may be obtained by the transaction.

When a transaction holds a lock that is no longer needed to ensure the transaction's serializability, or if a weaker level of isolation is acceptable, it is permissible to release the lock. The Concurrency Control Service therefore provides an operation that releases individual locks. This operation should be used with caution to ensure that the isolation level is appropriate for the application.

7.4 Nested Transactions

Lock conflicts within a transaction family are treated somewhat differently than conflicts between unrelated transactions. The underlying principle is the same for both: transactions must not be able to observe the effects of other transactions that might later abort. Unrelated transactions can abort independently; therefore, one transaction must not be permitted to acquire a lock that conflicts with a lock on the same resource held by an unrelated transaction.

Nesting imposes abort dependencies among related transactions. A parent transaction cannot abort without causing all of its children to abort. A child transaction that ends successfully cannot abort without causing its parent to abort. A transaction that cannot abort without causing another related transaction to abort (according to these guidelines and logical deductions) is said to be committed relative to that other transaction.

These dependencies make it possible to relax the rule that two transactions cannot acquire locks of conflicting modes on the same resource, without breaking the underlying principle. No partial effects can be observed and committed if all transactions that have done work cannot abort without the observer being aborted. This property translates into a sim-

2. See *Notes On Data Base Operating Systems* in *Operating Systems: An Advanced Course* (ed. Bayer, Graham, and Seegmuller) by J.N. Gray for further information.

ple rule for nested locking: if all transactions holding locks on a resource are committed with respect to a transaction trying to acquire a lock on the resource, no conflict exists.

The multiple possession model (see previous section) facilitates the use of locks with nested transactions. In this model, multiple related transactions may hold locks of conflicting modes on a resource at the same time. When a nested transaction requests a lock, it is granted if all of the transactions holding locks on the resource are committed relative to the requestor. Both the requestor and previous holders are then considered to hold locks on the resource.

A child transaction can acquire a lock on a resource locked by its parent and then drop that lock without causing its parent to lose its lock. A transaction cannot drop a lock that it did not acquire itself. The lock possession semantics also require that each transaction acquire locks on its own behalf. It is improper to take locks on behalf of another transaction or to depend on locks held by other transactions.

Other approaches to nested transactions[3] treat a resource as being locked by a single transaction at a time. When a nested transaction requests a lock on a resource that is already locked by an ancestor transaction, the nested transaction becomes the new owner of the lock. When a nested transaction commits, ownership of all of its locks is transferred to its parent. When a nested transaction aborts, ownership of its locks reverts to the previous owners. The Concurrency Control service performs these lock transfers automatically. The multiple possession semantics model is functionally equivalent to this model, but it supports simpler interfaces.

7.5 CosConcurrencyControl Module

The Concurrency Control Service is defined by the CosConcurrencyControl module, which provides interfaces that support both transactional and non-transactional modes of operation. This section defines the interfaces and describes the operations they support.

- The interfaces provide two modes of operation that correspond to those supported by the Transaction Service; in both modes, locks are identified by the lock set they are associated with and the mode of the lock.
- A client of the Concurrency Control Service may operate in an implicit mode such that locks are acquired on behalf of the current transaction (for transactional clients) or current thread (for non-transactional clients).
- For transactional clients, a second alternative is possible that involves the client identifying the transaction by means of a reference to the transaction's coordinator object (the explicit mode of operation).

Locks are acquired on lock sets. Two sets of operations are provided by the *LockSetFactory* interface to create lock sets, one creates a lock set that can be used by clients operating in the implicit mode (the *LockSet* interface), the other creates a lock set for explicit

3. See *Nested Transactions: An Approach To Reliable Distributed Computing* by J.E.B. Moss for further information.

mode transactional clients (the *TransactionalLockSet* interface). In addition, the *LockCoordinator* interface is provided to allow a client to release all locks held by a specific transaction.

The following sections define the types and exceptions common to both types of interface, the interfaces themselves, and describes the responsibilities of a user for managing transaction-duration locks.

OMG IDL for the CosConcurrencyControl module shown on the following page.

```
#include <CosTransactions.idl>
module CosConcurrencyControl {

    enum lock_mode {
        read,
        write,
        upgrade,
        intention_read,
        intention_write
    };

    exception LockNotHeld{};

    interface LockCoordinator
    {
        void drop_locks();
    };

    interface LockSet
    {
        void lock(in lock_mode mode);
        boolean try_lock(in lock_mode mode);

        void unlock(in lock_mode mode)
            raises(LockNotHeld);
        void change_mode(in lock_mode held_mode,
                    in lock_mode new_mode)
          raises(LockNotHeld);
        LockCoordinator get_coordinator(
            in CosTransactions::Coordinator which);
    };

    interface TransactionalLockSet
    {
        void lock(in CosTransactions::Coordinator current,
                in lock_mode mode);
        boolean try_lock(in CosTransactions::Coordinator current,
                    in lock_mode mode);
        void unlock(in CosTransactions::Coordinator current,
                in lock_mode mode)
          raises(LockNotHeld);
        void change_mode(in CosTransactions::Coordinator current,
                    in lock_mode held_mode,
                    in lock_mode new_mode)
```

```
        raises(LockNotHeld);
      LockCoordinator get_coordinator(
          in CosTransactions::Coordinator which);
   };

   interface LockSetFactory
   {
      LockSet create();
      LockSet create_related(in LockSet which);
      TransactionalLockSet create_transactional();
      TransactionalLockSet create_transactional_related(in
          TransactionalLockSet which);
   };
};
```

7.5.1 Types and Exceptions

The types and exceptions described in this section apply to both the *Lockset* and *TransactionalLockset* interfaces.

TABLE 2.

```
module CosConcurrencyControl {
   enum lock_mode {
      read,
      write,
      upgrade,
      intention_read,
      intention_write
   };

   exception LockNotHeld{};
```

lock_mode

The lock_mode type represents the types of lock that can be acquired on a resource.

LockNotHeld

The LockNotHeld exception is raised when an operation to unlock or change the mode of a lock is called and the specified lock is not held.

7.5.2 LockCoordinator Interface

The *LockCoordinator* interface enables a transaction service to drop all locks held by a transaction. The *LockSet* and *TransactionalLockSet* interfaces create instances of the *Lock-Coordinator* for each transaction. The *LockCoordinator* interface provides a single operation:

TABLE 3.

```
interface LockCoordinator {
    void drop_locks();
};
```

drop_locks

Releases all locks held by the transaction. This call is designed to be used by transactional clients when a transaction commits or aborts. For nested transactions, this operation must be called when the nested transaction aborts, but the call need only be made once for a transaction family when that family commits (recall that nested transaction commits are handled implicitly by the Concurrency Control service).

7.5.3 LockSet Interface

For clients operating in the implicit mode, locks are acquired and released on lock sets which are defined by means of the LockSet interface. The LockSet interface only provides operations to acquire and release locks on behalf of the calling thread or transaction. The interface does not provide support for transactional clients that use the explicit Transaction Service interfaces.

TABLE 4.

```
interface LockSet {
    void lock(in lock_mode mode);

    boolean try_lock(in lock_mode mode);

    void unlock(in lock_mode mode)
.       raises(LockNotHeld);

    void change_mode(in lock_mode held_mode,
                in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

When calls to acquire or release locks are made outside the scope of a transaction then it is assumed that the client is operating in the *non-transactional* mode (the concurrency control implementation must use the appropriate Transaction Service operation to determine whether the current thread is executing on behalf of a transaction).

lock

Acquires a lock on the specified lock set in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client then the operation will block the calling thread of control until the lock is acquired. If a call that is on behalf of a transactional client is blocked and the transaction is aborted then the call will return with the Transactions::TransactionRolledBack exception.

try_lock

Attempts to acquire a lock on the specified lock set. If the lock is already held in an incompatible mode by another client then the operation returns a FALSE result to indicate that the lock could not be acquired.

unlock

Drops a single lock on the specified lock set in the specified mode (recall that a lock can be held multiple times in the same mode). Calls to drop a lock that is not held result in the LockNotHeld exception being raised

change_mode

Changes the mode of a single lock (recall that multiple locks may be held on the same lock set). If the new mode conflicts with an existing mode held by an unrelated client, then the change_mode operation blocks the calling thread of control until the new mode can be granted. Like the lock call, if the client is a transaction and it aborts while the thread of control if blocked then the Transactions::TransactionRolledBack exception will be raised. Similarly, when a call is made to change the mode of a lock, but the lock is not held in the specified mode, the LockNotHeld exception will be raised.

get_coordinator

Returns the lock coordinator associated with the specified transaction.

7.5.4 TransactionalLockSet Interface

The *TransactionalLockSet* interface provides operations to acquire and release locks on a lock set on behalf of a specific transaction. The operations that make up the *Transactional-*

LockSet interface are:

TABLE 5.

```
interface TransactionalLockSet {
    void lock(in CosTransactions::Coordinator which,
              in lock_mode mode);

    boolean try_lock(in CosTransactions::Coordinator which,
                     in lock_mode mode);

    void unlock(in CosTransactions::Coordinator which,
                in lock_mode mode)
        raises(LockNotHeld);

    void change_mode(in CosTransactions::Coordinator which,
                     in lock_mode held_mode,
                     in lock_mode new_mode)
        raises(LockNotHeld);

    LockCoordinator get_coordinator(in
        CosTransactions::Coordinator which);
};
```

The operations provided by the *TransactionalLockSet* interface operate in an identical manner to the equivalent operations provided by the *LockSet* interface. The interfaces differ in that for the *TransactionalLockSet* interface the identity of the transaction is passed explicitly as a reference to the coordinator for the transaction instead of implicitly through an association with the calling thread.

7.5.5 LockSetFactory Interface

Lock sets are created using the *LockSetFactory* interface.

TABLE 6.

```
interface LockSetFactory {
    LockSet create();
    LockSet create_related(in LockSet which);

    TransactionalLockSet create_transactional();
    TransactionalLockSet
        create_transactional_related(in
            TransactionalLockSet which);
};
```

This interface provides two sets of operations that return new *LockSet* and *Transactional-LockSet* instances.

create

Creates a new lock set and lock coordinator.

create_related

Creates a new lock set that is related to an existing lock set. Related lock sets drop their locks together.

create_transactional

Creates a new transactional lock set and lock coordinator for explicit mode transactional clients.

create_transactional_related

Creates a new transactional lock set that is related to an existing lock set. Related lock sets drop their locks together.

Externalization Service Specification 8

Contents

This chapter contains the following topics.

Topic	Page
“Service Description”	8-2
“Service Structure”	8-2
“Object and Interface Hierarchies”	8-8
“Interface Summary”	8-11
“CosExternalization Module”	8-13
“CosStream Module”	8-16
“Specific Externalization Relationships”	8-26
“The CosExternalizationContainment Module”	8-27
“The CosExternalizationReference Module”	8-28
“Standard Stream Data Format”	8-29
“References”	8-31

Note – Dec. 1998: OMG made some editorial changes. These changes involved merging the old CosCompoundExternalization section and IDL into the CosStream section and IDL. Those were the only changes, along with a few text references to CosCompoundExternalization that became CosStream, which have been marked with changebars.

8.1 Service Description

The Externalization Service specification defines protocols and conventions for externalizing and internalizing objects. To externalize an object is to record the object's state in a stream of data. Objects which support the appropriate interfaces and whose implementations adhere to the proper conventions can be externalized to a stream (in memory, on a disk file, across the network, etc.) and subsequently be internalized into a new object in the same or a different process. The externalized form of the object can exist for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

Many different externalized data formats and storage mediums can be supported by service implementations. But, for portability, clients can request that externalized data be stored in a file using a standardized format that is defined as part of this Externalization Service specification.

Externalizing and internalizing an object is similar to copying the object. The copy operation creates a new object that is initialized from an existing object. The new object is then available to provide service. Furthermore, with the copy operation, there is an assumption that it is possible to communicate via the ORB between the "here" and "there". Externalization, on the other hand, does not create an object that is initialized from an existing object. Externalization "stops along the way". New objects are not created until the stream is internalized. Furthermore, there is no assumption that is possible to communicate via the ORB between "here" and "there."

The Externalization Service is related to the Relationship Service. It also parallels the Life Cycle Service in defining externalization protocols for simple objects, for arbitrarily related objects, and for graphs of related objects that support compound operations. (For more information, refer to the Service Dependencies section in Chapter 2.)

The Externalization Service defines protocols in these areas:

- Client's view of externalization, composed of the interfaces used by a client to externalize and internalize objects. The client's view of externalization is defined by the **Stream** interface.
- Object's view of externalization, composed of the interfaces used by an externalizable object to record and retrieve their object state to and from the stream's external form. The object's view is defined by the **StreamIO** interface.
- Stream's view of externalization, composed of the interfaces used by the stream to direct an externalizable object or graph of objects to record or retrieve their state from the stream's external form. The stream's view of externalization is given by the **Streamable**, **Node**, **Role** and **Relationship** interfaces.

8.2 Service Structure

This section explains the model of externalization for client and stream. It also describes the model of externalization and internalization for objects.

8.2.1 Client's Model of Object Externalization

A client has a simple view of the externalization service. A client that wishes to externalize an object first must have an object reference for a **Stream** object. A **Stream** object owns and provides access to the externalized form of one or more objects. Streams may be provided that hold externalized data on various mediums such as in memory or on disk. All Externalization Service implementors provide a **Stream** object that saves the externalized data in a file. A client may create a **Stream** object using the **create()** operation on a **StreamFactory** object, or may specify that a file be used to store the externalized data using the **create()** operation of a **FileStreamFactory** object.

The client can create a **Stream** object that supports a standardized externalization data format. Externalization data that follows this format will be internalizable on all CORBA-compliant ORBs that can locate compatible object implementations. By including support for a specific external representation format in the Externalization Service, portability of object state is provided across different CORBA-compliant implementations and hardware architectures.

Once a client has a **Stream** object, the client may externalize an object by issuing an **externalize()** request on the **Stream** object, providing the object reference to the object that should be externalized. In general, the client is unaware of whether externalizing an object causes any other related objects to be externalized. An externalizable object may represent a simple object, a set of objects, or a graph of related objects. The client uses the same interface in all cases.

If a client wishes to externalize multiple objects (or related sets of objects) to the same stream, the client issues a **begin_context()** request before the first externalize request and then issues an **end_context()** following the last externalize request for that same stream.

The externalized form of the object can exist in the stream object for arbitrary amounts of time, be transported by means outside of the ORB, and can be internalized in a different, disconnected ORB.

A client that wishes to internalize an object issues an **internalize()** request on the appropriate **Stream** object, providing a factory finder. The **Stream** object interacts with the specified factory finder, or uses other implementation dependent mechanisms, to create an implementation of the object that matches the externalized data. The client is returned an object reference to the newly internalized object.

8.2.2 Stream's Model of Object Externalization

A **Stream** object provides the **Stream** interface for use by clients. The **Stream** object is also responsible for providing an object that supports a **StreamIO** interface for actually reading and writing data to the externalized data form. The stream object may support the **StreamIO** interfaces itself, or may create another object that supports the **StreamIO** interfaces. This is considered an implementation detail.

Note – When the behavior described in this section may be implemented in either the **Stream** or **StreamIO** objects (or other internal objects they may use), the term "stream service" is used.

When a stream object receives an externalize request from a client, it also gets an object reference to the object to be externalized. The stream cooperates with the externalizable object to accomplish externalization and internalization, using the object's **Streamable** interfaces.

The stream service uses the readonly **Key** attribute of the externalizable object to decide what information to put into the external data in order to be able to find the correct factory and implementation with which to subsequently internalize an equivalent object. The stream service then issues an **externalize_to_stream()** request to the externalizable object, providing an object reference to a **StreamIO** object that is to be used by the externalizable object to record its state in the stream service's external data.

When a **Stream** object receives an internalize request from a client, it also gets a factory finder. The stream service holds the external form of the object, or set of objects, to be internalized. The stream service reads the key from its externalized data. It may then pass the key to the factory finder to locate a factory that can create an object with an implementation that matches the recorded object state. The stream service implementation may use other implementation specific ways of creating an appropriate object. The stream service then issues an **internalize_from_stream()** request to the newly created object, providing an object reference to a **StreamIO** object that is used by the externalizable object to initialize its state according to the stream service's externalized data.

When a **Stream** object receives a **begin_context()** request, the stream service sets up a context during which the stream service ensures that externalizing multiple objects that may have overlapping object references and/or object relationships produces single instances of those objects on internalization. An **end_context()** request causes the stream service to remove the previous internal context, and externalize subsequent objects without regard to whether they have already been externalized in this **Stream**'s data.

8.2.3 Object's Model of Externalization

Every object that wishes to be externalizable must support the **Streamable** interface, and follow conventions on use of the **StreamIO** interfaces to record and retrieve their object state from a **Stream**'s data.

When a **Streamable** object receives an **externalize_to_stream** request from the stream service, it must write all of its state necessary for internalization to the **StreamIO** object provided by the stream service. **StreamIO** provides **write_<type>()** operations for writing each of the CORBA basic data types, plus string types. If an object has object references that are part of its state, the **StreamIO** **write_object()** operation may be used to cause the object specified by an object reference to also be externalized to the stream's data.

Externalization Control Flow (streamable object is not a node)

Client calls **Stream::externalize(Streamable object)**

Stream writes a key for this object to the external representation.

Stream calls the **Streamable::write_to_stream(StreamIO this_sio)** so that the object can write out whatever internal state it needs to save.

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 8-2

Streamable object writes out its non-object data using the primitive **StreamIO::write_...**(data) functions

Streamable object writes out other objects using the **StreamIO::write_object(Streamable object)** function

Figure 8-1 Externalization control flow when streamable object is not in a graph of related objects

A streamable object may be a node in a graph of related objects, that is, it may use the Relationship Service to connect to other objects and support the **CosStream::Node** interface. Such a streamable object simply delegates the **Streamable::externalize_to_stream()** request back to the stream service, using the **StreamIO::write_graph()** operation.

The stream service then coordinates the externalization of the graph and calls the object back using the object's **CosStream::Node** interface.

Externalization Control Flow (streamable is a node)

Streamable object, recognizing that it is a node in a graph of related objects, delegates the externalization of the graph to the stream service using **StreamIO::write_graph**(this_node) operation.

StreamIO::write_graph,coordinates the externalization of the graph using **Node::externalize_node(this_sio)** operation.

Node writes out its non-object data using the primitive **StreamIO::write_...(data)** functions

Node writes out other objects using the **StreamIO::write_object(Streamable object)** function

Node writes out its role objects using the **Role::externalize_role(this_sio)** operation.

StreamIO::write_graph uses propagation value to determine next nodes and writes a key for next node

StreamIO object externalizes the involved relationships using **Relationship::externalize()**. **StreamIO** writes traversal scoped ids for the externalized roles and relationships to the **Stream**'s data.

Figure 8-2 Externalization control flow when streamable object is a node in a graph of related objects

8.2.4 Object's Model of Internalization

When a streamable object receives an **internalize_from_stream()** request from a stream, it must read data from the **StreamIO** object provided by the stream service, and initialize its state to match the externalized state. The externalizable object requests data from the stream service using the **StreamIO read_<type>()** operations for basic data, and string types. If the object being internalized includes a reference to another object as part of its state, the **StreamIO read_object()** operation may be used to have that object also internalized from the stream's data.

Internalization Control Flow (streamable object is not a node)

Client calls **Streamable = Stream::internalize(FactoryFinder f)**

Stream reads key from the external representation, and uses this and the factory finder to *create an object* of the correct interface and implementation. The stream may use the **StreamableFactory** interface.

Stream calls the **Streamable::read_from_stream(StreamIO this_sio)** so that the object can read the data in its external representation and reset or calculate its internal state

If **Streamable** object is a node in a graph of related objects, flow is given in Figure 8-4

Streamable object reads in its non-object data using the primitive **StreamIO::read_...(data)** functions

Streamable object internalizes other objects using the **Streamable = StreamIO::read_object()** function

Figure 8-3 Internalization control flow when object is not in a graph of related objects

A streamable object may be a node in a graph of related objects, that is, it may use the Relationship Service to connect to other objects and support the **CosStream::Node** interface. Such a streamable object simply delegates the **Streamable::internalize_from_stream()** request back to the stream service, using the **StreamIO::write_graph()** operation.

The stream service then coordinates the externalization of the graph and calls the object back using the object's **CosStream::Node** interface.

Internalization Control Flow (streamable is a node)

Streamable object, recognizing that it is a node in a graph of related objects, delegates the internalization of the graph to the stream service using **StreamIO::read_graph(this_node)** operation.

StreamIO::read_graph,coordinates the internalization of the graph using **Node::internalize_node(this_sio)** operation.

Node reads its non-object data using the primitive **StreamIO::read_...(data)** functions

Node read other objects using the **StreamIO::read_object(Streamable object)** function

Node reads its role objects using the **Role::internalize_role(this_sio)** operation.

StreamIO::read_graph reads the key for next node and uses the StreamableFactory interface to create the next node.

StreamIO object internalizes the traversal scoped identifiers for the externalized roles and relationships and internalizes the relationships using **Relationship::internalize()**.

Figure 8-4 Internalization control flow when object is in a graph of related objects

8.3 Object and Interface Hierarchies

This section identifies the objects required for the Externalization Service and important inheritance and use relationships that exist between their interfaces.

The Object Externalization Service can only externalize and internalize objects that inherit the **Streamable** interface. **Streamable** does not inherit any other interfaces. However, it must have an associated **StreamableFactory** that the Externalization Service implementation can find and use when internalizing the object.

Stream inherits the **LifeCycleObject** interface because clients of the Externalization Service need to remove these objects. The **StreamFactory** or **File StreamFactory** interfaces may be used to create stream objects.

In addition to the inheritance relationships described above, the class diagram in Figure 1 also shows the usage relationships between the service objects. **Stream externalize()** and **internalize()** operations invoke the **Streamable externalize_to_stream()** and **internalize_from_stream()** operations to write and read the appropriate object internal state. A **StreamIO** object is passed as an argument to these operations. The externalized object determines how much of its state must be put in the external representation, and can minimize saved state by recreating some state upon internalization. The **Streamable externalize_to_stream()** and **internalize_from_stream()** use **StreamIO** operations to actually put various data types and contained object references in the external representation. This allows **StreamIO** to put appropriate headers in the external representation so that the object can be recreated correctly during internalization. The **Stream** is responsible for providing an object that supports the **StreamIO** interface. The **Stream** object may support the **StreamIO** interface itself, or create another object that supports the **StreamIO** interface. The **Stream** and **StreamIO** implementations decide on the storage medium and data type representation conversion for different hardware, without requiring different implementation of the objects being externalized.

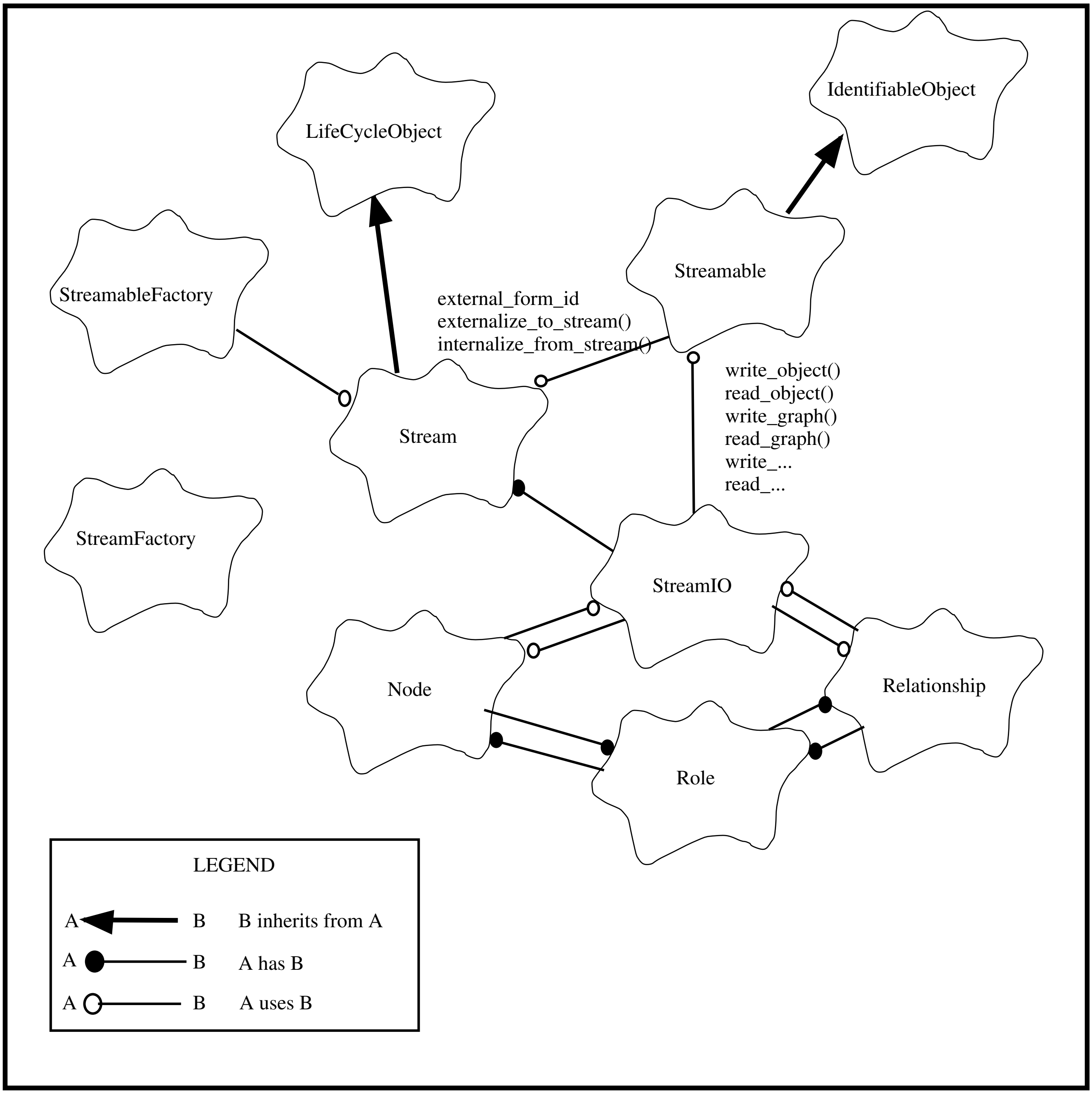


Figure 8-5 Object Externalization Service Booch Class (=Interface) Diagram

8.4 Interface Summary

The Externalization Service defines interfaces (using OMG IDL) to support the functionality described in the previous sections. The following tables give high level descriptions of the Externalization Service interfaces. Subsequent sections describe the interfaces in more detail.

Table 8-1 Client Functional Interfaces support client's model of externalization

Interface	Purpose	Primary Client
Stream	Holds external form of objects.	Clients that need to externalize and internalize objects.
StreamFactory	Creates and initializes stream objects.	Clients that need to create stream objects.
FileStreamFactory	Creates and initializes stream objects that stores data in a file.	Clients that need to create stream objects, and want the externalized data in a file.

Table 8-2 Service Construction Interfaces support service implementation's model of externalization

Interface	Purpose	Primary Client
Streamable	Provides its state to a stream for externalization, and gets its state from the stream on internalization.	The stream service implementation of externalization and internalization.
StreamableFactory	Creates and initializes streamable objects	The stream service internalization implementation.
StreamIO	Part of stream implemenation that writes and reads object state to appropriately converted external form.	The externalizable objects that need to record and retrieve their state from a stream.

Table 8-3 Compound Externalization Interfaces support service implementation's model of graph externalization

Interface	Purpose	Primary Client
Node	Defines externalization and internalization operations on nodes in graphs of related objects.	The stream service implementation of externalization and internalization.
Relationship	Defines externalization and internalization operations on relationships.	The stream service implementation of externlization and internalization.
Role	Defines externalization and internalization operations on roles.	The stream service implementation of externalization and internalization.

8.4.1 Externalization Service Architecture: Audience/Bearer Mapping

Stream and **StreamFactory** are solely functional interfaces. Their audience is the client of the Externalization Service.

Streamable, **StreamableFactory**, and **StreamIO** are solely construction interfaces. The audience for **Streamable** is both the **Stream** and **StreamIO** objects. To be "externalizable," objects must inherit the **Streamable** interface and provide implementations of its operations. The audience for **StreamIO** interface is the externalizable **Streamable** and **StreamableNode** objects. The **StreamIO** objects are part of the Externalization Service implementation.

The **Stream**, **StreamFactory**, and **StreamIO** objects are specific objects because their purpose is to provide a part of the Externalization Service. However, there may be many **Stream** and **StreamIO** instances in a system, since each represents a particular external representation of an object or group of objects.

Streamable and **StreamableFactory** objects are generic objects because their primary purpose is unrelated to the Externalization Service. Any definer or implementor of an object may choose to inherit the **Streamable** interface in order to support externalization/internalization of that object.

In summary:
- **Stream** and **StreamFacto**ry are specific functional interfaces
- **Streamabl**e and S**treamableFactory** are generic construction interfaces
- **StreamIO** is a specific construction interface

8.5 CosExternalization Module

The client-functional interfaces defined by the the CosExternalization module are:

- **StreamFactory** interface, which creates a stream.
- **FileStreamFactory** interface, which has an operation that lets clients cause externalized data be stored in a file or internalize objects from a file they have been given.
- **Stream** interface, which can externalize one object or a group of objects; finalize the externalization, and internalize an object.

```
/File: CosExternalization.idl
//Part of the Externalization Service

#ifndef _COS_EXTERNALIZATION_IDL_
#define _COS_EXTERNALIZATION_IDL_

#include <CosLifeCycle.idl>
#include <CosStream.idl>

#pragma prefix “omg.org”

module CosExternalization {
    exception InvalidFileNameError{};
    exception ContextAlreadyRegistered{};
    interface Stream: CosLifeCycle::LifeCycleObject{
        void externalize(
            in CosStream::Streamable theObject);
        CosStream::Streamable internalize(
            in CosLifeCycle::FactoryFinder there)
            raises( CosLifeCycle::NoFactory,
                    CosStream::StreamDataFormatError );
        void begin_context()
            raises( ContextAlreadyRegistered);
        void end_context();
        void flush();
    };
    interface StreamFactory {
        Stream create();
    };
    interface FileStreamFactory {
        Stream create(
            in string theFileName)
            raises( InvalidFileNameError );
    };
};
#endif /* ifndef _COS_EXTERNALIZATION_IDL_ */
```

8.5.1 StreamFactory Interface

8.5.1.1 Creating a Stream Objec

Stream create();

Clients of the Object Externalization Service must create a **Stream** object before they can externalize or internalize any objects. Two factory interfaces are supported. The first, the **StreamFactory** interface has a **create()** operation that creates a stream without specifying any special characteristics of the implementation.

8.5.2 FileStreamFactory Interface

8.5.2.1 Creating a Stream Objec Associated with a File

```
Stream create(
    in string theFileName)
    raises( InvalidFileNameError );
```

For clients that want to cause the externalized data stored in a file, or that need to internalize objects from a file they have been given, the **FileStreamFactory** interface has a **create()** operation that takes a string input parameter. The client sets this string to the filename of the file that will be used by the stream service to hold the externmal representation of the objects externalized, or that contains the external representation of objects that the client wishes to internalize. **Stream::externalize()** requests will append to any existing data in the file associated with a stream.

8.5.3 Stream Interface

8.5.3.1 Externalizing an Object

```
void externalize(
    in CosStream::Streamable theObject);
```

Clients of the Object Externalization Service invoke **externalize()** on a **Stream** object passing the object reference of a **CosStream::Streamable** object, **theObject**, to be externalized. Only objects that are of type **CosStream::Streamable** can be externalized. Subsequently, clients invoke the **internalize()** operation on the **Stream** containing the external representation, and **Stream internalize()** operation creates a new object with state identical to what was externalized and returns the new object reference.

The implementation of **externalize()** writes implementation specific header information to the external representation it is maintaining, so that the correct object can be recreated at internalization time. This could be the factory key that was used to create the **CosStream::Streamable** object, or could include the interface type, implemenation repository, or factory object names. The factory key may be obtained by from the **external_form_id** attribute of **theObject**. The **externalize()**

implementation must then invoke the **CosStream::Streamable externalize_to_stream()** operation on **theObject** to cause the object's internal state to be written to the external respresentation. The **Stream** is responsible for providing an object that supports the **StreamIO** interfaces for the externalizable object to use in writing data to the stream service.

8.5.3.2 Externalizing Groups of Objects

void begin_context()
raises(ContextAlreadyRegistered);
void end_context();

If a client wishes to externalize a set of objects with overlapping references and/or object relationships, the client invokes **begin_context()** on the **Stream**. This must be called before externalizing any of the set of objects, and **end_context()** must be called on the **Stream** after the entire set of objects has been externalized and before the **Stream** is used with another set of objects.

The **Stream** implementation establishes an association with the specified **Stream** object and a logical "context". The **Stream** ensures that all objects externalized to this stream while this association lasts will be externalized in such a way that internalization will not create any duplicate objects. That is, the implementation of **Stream** checks for "context", and for objects externalized in the same context handles overlapping or circular references and/or relationships between those objects. The association lasts until **end_context()** is called. The **Stream** raises the **ContextAlreadyRegistered** exception if **begin_context()** is called and a context is already established, perhaps through some other implementation dependent mechanism or perhaps because **end_context()** has not been called following a previous **begin_context()**.

8.5.3.3 Completing Externalization

void flush();

Clients invoke **flush()** to request that the external representation is committed to its final storage medium, whatever that may be. The implementation of **flush()** should attempt to ensure that the external respresentation is completely up-to-date in its final storage (e.g. memory buffer, file, tape, ...).

8.5.3.4 Internalizing an Object

CosStream::Streamable internalize(
in CosLifeCycle::FactoryFinder there)
raises(CosLifeCycle::NoFactory,
CosStream::StreamDataFormatError);

The implementation of **internalize()** must create an object with the correct interface and implementation to match the externalized representation and return a pointer to the new **CosStream::Streamable** object. The **internalize()** implementation must then invoke the **internalize_from_stream()** operation on the new object. The

CosStream::StreamDataFormatError exception should be raised if an error is detected in the data format of the object header. The **CosLifeCycle::NoFactory** exception should be raised if the object cannot be created because an appropriate factory cannot be found. If the object cannot be created due to other reasons, an ObjectCreationError exception should be raised. Additional **CosStream::StreamDataFormat** exceptions may be raised by the **read_<type>** operations invoked by **internalize_from_stream()** operation due to errors in the externalized data format.

8.6 CosStream Module

The service construction interfaces defined by the CosStream module are:

- **Streamable** interface
- **StreamableFactory** interface
- **StreamIO** interface

If a **Streamable** object participates as a node in a graph of related objects, the *Streamable* object can delegate the externalization operation to the stream service. In particular, the **Streamable** object simply uses the **write_graph()** operation. The **write_graph()** operation expects a streamable object reference as a starting node. The stream service narrows the streamable object reference to **CosStream::Node**. The **write_graph()** then coordinates the orderly externalization of the graph of related objects. For more details on compound operations, see the Relationship Service specification and the Compound Life Cycle section in the Life Cycle Service specification.

The CosStream module defines interfaces for use by the **write_graph()** operation.*:*

- **Node** interface
- **Role** interface
- **Relationship** interface
- **PropagationCriteriaFactory** interface

```
//File: CosStream.idl
//Part of the Externalization Service
// Modified from version 1.0 to include the previous CosCompoundExternal-
ization module

#ifndef _COS_STREAM_IDL_
#define _COS_STREAM_IDL_

#include <CosLifeCycle.idl>
#include <CosObjectIdentity.idl>
#include <CosGraphs.idl>

#pragma prefix “omg.org”

module CosStream {
    exception ObjectCreationError{};
    exception StreamDataFormatError{};
```

```
interface StreamIO;
interface Node;
interface Role;
interface Relationship;

interface Streamable:
  CosObjectIdentity::IdentifiableObject {
      readonly attribute CosLifeCycle::Key external_form_id;
      void externalize_to_stream(
            in StreamIOtargetStreamIO);
      void internalize_from_stream(
            in StreamIOsourceStreamIO,
            in CosLifeCycle::FactoryFinder there)
            raises(CosLifeCycle::NoFactory,
                        ObjectCreationError,
                        StreamDataFormatError );
};

interface StreamableFactory {
      Streamable create_uninitialized();
};


interface StreamIO {
      void write_string(in string aString);
      void write_char(in char aChar);
      void write_octet(in octet anOctet);
      void write_unsigned_long(
            in unsigned long anUnsignedLong);
      void write_unsigned_short(
            in unsigned short anUnsignedShort);
      void write_long(in long aLong);
      void write_short(in short aShort);
      void write_float(in float aFloat);
      void write_double(in double aDouble);
      void write_boolean(in boolean aBoolean);
      void write_object(in Streamable aStreamable);
      void write_graph(in Node aNode);
      string read_string()
            raises(StreamDataFormatError);
      char read_char()
            raises(StreamDataFormatError );
      octet read_octet()
            raises(StreamDataFormatError );
      unsigned long read_unsigned_long()
            raises(StreamDataFormatError );
      unsigned short read_unsigned_short()
            raises( StreamDataFormatError );
      long read_long()
            raises(StreamDataFormatError );
```

```
        short read_short()
            raises(StreamDataFormatError );
        float read_float()
            raises(StreamDataFormatError );
        double read_double()
            raises(StreamDataFormatError );
        boolean read_boolean()
            raises(StreamDataFormatError );
        Streamable read_object(
            in CosLifeCycle::FactoryFinder there,
            in Streamable aStreamable)
            raises(StreamDataFormatError );
        void read_graph(
            in Node starting_node,
            in CosLifeCycle::FactoryFinder there)
            raises(StreamDataFormatError );
    };

// the following are required for compound externalization

    struct RelationshipHandle {
        CosRelationships::Relationship theRelationship;
        CosObjectIdentity::ObjectIdentifier constantRandomId;
    };

    interface Node : CosGraphs::Node, CosStream::Streamable{
        void externalize_node (in CosStream::StreamIO sio);
        void internalize_node (in CosStream::StreamIO sio,
            in CosLifeCycle::FactoryFinder there,
            out Roles rolesOfNode)
                raises ( CosLifeCycle::NoFactory);
    };

    interface Role : CosGraphs::Role {
        void externalize_role (in CosStream::StreamIO sio);
        void internalize_role (in CosStream::StreamIO sio);
        CosGraphs::PropagationValue externalize_propagation (
            in RelationshipHandle rel,
            in CosRelationships::RoleName toRoleName,
            out boolean sameForAll);
    };

    interface Relationship : CosRelationships::Relationship {
        void externalize_relationship (
            in CosStream::StreamIO sio);
        void internalize_relationship(
            in CosStream::StreamIO sio,
            in CosGraphs::NamedRoles newRoles);
        CosGraphs::PropagationValue externalize_propagation (
        in CosRelationships::RoleName fromRoleName,
        in CosRelationships::RoleName toRoleName,
```

```
        out boolean sameForAll);
    };

    interface PropagationCriteriaFactory {
        CosGraphs::TraversalCriteria create_for_externalize( );
    };


};
#endif  /* ifndef _COS_STREAM_IDL_  */
```

Since IDL only supports template instantiations rather than templates themselves, the fixed-point decimal template type cannot be used directly for the **write_fixed** and **read_fixed** operations. Instead, the **fixed** type instances must be passed to and from these routines as **any**s with **TypeCode**s of **tk_fixed**.

8.6.1 Standard Stream Data Format

The standard stream format for each new IDL type is shown in the table below. Also shown are the standard formats for types **char** and **string**, which have been extended to state explicitly that data is encoded as defined by ISO 8859-1.

Tag	CORBA Type	Data Format
x'F1'	char	one byte, encoded as defined by ISO 8859-1
x'FA'	string	null-terminated sequence of bytes, encoded as defined by ISO 8859-1
x'E1'	char	an unsigned long code set tag, followed by a one byte data value, encoded as defined by code set tag
x'E2'	string	an unsigned long code set tag, followed by a null-terminated sequence of characters, encoded as defined by code set tag
x'E3'	fixed<d,s>	an unsigned short byte count (d+2)/2), followed by (d+2)/2 bytes in CDR format.
x'FE'	wchar	an unsigned long code set tag, followed by a data value, encoded as defined by code set tag
x'FF'	wstring	an unsigned long code set tag, followed by a null-terminated sequence of wchar, encoded as defined by code set tag
x'FB'	long long	eight bytes, big-endian format
x'FC'	unsigned long long	eight bytes, big-endian format
x'FD'	long double	sixteen bytes, IEEE 754 format, sign bit in first byte

The first two entries in the table describe the current formats for **char** and **string**, modified only to state explicitly, rather than implicitly, that the encoding used is defined by ISO 8859-1. These existing formats are unchanged for backward compatibility purposes.

The next two entries (x'E1' and x'E2') define tagged formats for **char** and **string**, which consist of a code set tag (from the OSF Character and Code Set Registry) followed by an actual data value. The motivation for these tagged formats is to prevent information loss, which may occur for some native code sets when converted to ISO 8859-1 (i.e., when such data is externalized in the formats described in the first two entries). However, if character and string data is externalized in a form other than ISO 8859-1, some ORBs may not be able to internalize it successfully (e.g., because an appropriate converter is not available), thus reducing the portability of the externalized data. So, if maximum portability is desired, character and string data should be externalized in ISO 8859-1 form.

The remaining entries in the table describe the formats for the new IDL types. Note that the previous discussion about the tradeoff between portability and information loss for externalized character and string data also applies to wide character and wide string data. If maximum portability is desired, wide character and wide string data should be externalized in Unicode form, while if using this form would result in an unacceptable loss of information, then a form other than Unicode should be used.

Data values of type **wchar** and **wstring** are represented as one or more octets, or an unsigned integer, depending on the code set used. This is similar to the on-the-wire representation of **wchar** and **wstring** data.

8.6.2 The StreamIO Interface

The **write_<type>()** and **read_<type>()** operations on **StreamIO** are used by **Streamable externalize_to_stream()** and **internalize_from_stream()** operations to cause internal object state to be written to or read from the external representation. The **externalize_to_stream()** decomposes the internal state of an object in a series of primitive data type values that can be written and read with these operations. **StreamIO** supports writing and reading all the CORBA basic data types.

The implementation of the **write_**... and **read_**... operations are responsible for any desired conversion of the data and transfering the data to or from the desired external representation. Actual transfer of the representation to the final storage medium may be deferred until the **flush()** operation. All details of the external representation format, storage medium, and buffering are specific to the implementation. Different implementations may support buffering of the external representation data in memory, converting data values to a canonical binary form for exchange across big/little endian CPU hardware, conversion of data to a canonical text form for readability or to facilitate mailing objects across networks, use of various storage mediums such as memory, filesystem, tape or other differences. See "Standard Stream Data Format" on page 8-19 for information on a portable external representation. A **StreamDataFormatError** exception should be raised if errors are detected in the data format of the external representation.

In support of integrating the Externalization Service with the Transaction and Persistent Object Services, the **read_object** operation supports the internalization to existing objects. The semantics of the operation are that if the **aStreamable** parameter is Null, then the **FactoryFinder** parameter is used to create an instance for internalize. If the **aStreamable** parameter is not Null, then the **StreamIO** implementation will internalize to a streamable object. This semantic allows the Externalization Service to be used as a Persistent Object Service protocol and to support the restore operation on existing objects in the case of an aborted transaction.

8.6.3 The Streamable Interface

Object implementors must inherit from the **Streamable** interface if they want an object to be externalizable. Three operations must be implemented.

Comparing Streamable Objects

boolean CosObjectIdentity::IdentifiableObject::is_identical(
in CosObjectIdentity::IdentifiableObject anObject);
readonly unsigned long constant_random_id;

A **Streamable** object inherits from **CosObjectIdentity::IdentifiableObject**, and therefore must support a **constant_random_id** attribute and an **is_identical()** operation. The stream service uses these to compare objects when detecting cycles or overlapping references in objects being externalized to the same stream in the same context or within the same graph. The **constant_random_id** attribute value does not have to be unique, but a unique value may substantially speed up the externalization process.

Creation Key for a Streamable Object

readonly attribute CosLifeCycle::Key external_form_id;

An **Streamable** object must support a readonly attribute, **external_form_id**, which is a key that can be given to a factory finder in order to find a factory that could have created this object. The stream service may use this attribute during internalization to create an object that can reinitialize itself from the externalized data.

8.6.3.1 Writing the Object's State to a Stream

void externalize_to_stream(
in StreamIOtargetStreamIO);

The **externalize_to_stream()** operation is responsible for decomposing an externalizable object's internal state into a series of primitive data type values and object references. The **externalize_to_stream()** function must write out all the neccessary primitive data values using the **write_<type>()** operations on the **targetStreamIO** for non-object data types. If this object has other object references, then, normally, those objects should also be written out using the **write_object()** operation on the **targetStreamIO**. However, it is up to the **Streamable** implementor to decide which referenced objects should be externalized with this object. The primitive data values must all be written before any of the embedded objects references are written.

If the **Streamable** is a node in a graph, then it should delegate the **externalize_to_stream()** to the **StreamIO** by invoking **write_graph()**. The object would subsequently receive an **externalize_node_to_stream()** and write out its internal state as described above. **Node** objects should not call **write_object()** for other nodes in their graph, but may call **write_object()** for object references that are not for nodes in their graph.

8.6.3.2 *Reinitializing the Object's State from a Stream*

```
void internalize_from_stream(
    in StreamIOsourceStreamIO,
    in CosLifeCycle::FactoryFinder there)
    raises(CosLifeCycle::NoFactory,
           ObjectCreationError,
           StreamDataFormatError );
```

The **internalize_from_stream()** operation is responsible for reinitializing the object's internal state from the series of primitive data type values and object references that are written/flattened during **externalize_to_stream()**. The **internalize_from_stream()** operation should read in all the neccessary internal state of the object using the **read_<type>()** operations on the **sourceStreamIO** for non-object data types. If this object has other object references that were externalized using **write_object()**, then those objects should be recreated using the **read_object()** operation on the **sourceStreamIO** with the same **FactoryFinder** argument as the **there** parameter passed in to the **internalize_from_stream()** operation. The **read_<type>()** and **read_object()** operations for the various portions of the object's internal state must be invoked in the same order in which they are written by the **externalize_to_stream()** implementation. The **internalize_from_stream()** must also initialize any additional state that was not externalized because it can be derived from other state information. Therefore, the **externalize_to_stream()** and **internalize_from_stream()** operations must be designed to complement each other.

If the **Streamable** is a node in a graph, then it should delegate the **internalize_to_stream()** to the **sourceStreamIO** by invoking **read_graph()** with the same **FactoryFinder** argument as the **there** parameter passed in to the **internalize_from_stream()** operation. The **Streamable** (also **Node**) object would subsequently receive an **internalize_node_to_stream()** and read in its internal state as described above. **Node** objects should not call **read_object()** for other nodes in their graph, but may call **read_object()** for object references that are not for nodes in their graph..

The ObjectCreationError and StreamDataFormatError exceptions originate from the **read_object()** and **read_<type>** operations on the **sourceStreamIO**, and are not explicitly raised by the **internalize_from_stream()** code.

8.6.4 *The StreamableFactory Interface*

8.6.4.1 *Creating a Streamable Object*

Streamable create_uninitialized();

The stream service must be able to create a **Streamable** object in order to internalize an object from the stream's externalized data. For any externalizable object, a **StreamableFactory** object must exist that supports creation of that object. This factory must be findable using the **readonly external_form_id Key attribute** of the streamable object. The stream service implementation could store this key during externalization and use it during internalization to find the factory that can create the externalized object. However, a stream implementation may use other means to create the object during internalization. The **create_uninitialized()** operation on the **StreamableFactory** should create the associated streamable object. This streamable object does not have to be initialized, since that can be done on the subsequent **internalize_from_stream()** operation on the newly created streamable object.

8.6.5 The Node Interface

The **Node** interface defines operations to internalize and externalize a node.

8.6.5.1 Externalizing a Node

void externalize_node (in CosStream::StreamIO sio);

The **externalize_node()** operation transfers the node's state to the stream given by the **sio** parameter. The node is responsible to externalize its roles as well. The node can accomplish this by writing the role's key to the stream and using the **Role::externalize_role()** operation.

8.6.5.2 Internalizing a Node

```
void internalize_node (in CosStream::StreamIO sio,
    in CosLifeCycle::FactoryFinder there,
    out Roles rolesOfNode)
        raises ( CosLifeCycle::NoFactory);
```

The **internalize_node()** operation causes a node and its roles to be internalized from the stream **sio**.

It is the node's responsibility to create and internalize its roles. It can do this by reading the key for a role from the stream and using the **CosStream::StreamableFactory** interface to create the uninitialized role and the **CosStream::internalize_role()** operation to internalize the role. The new roles should be collocated with the factory finder given by the **there** parameter.

The result of a **internalize_node()** operation is a sequence of roles.

Figure 8-6 illustrates the result of an internalize. A node, when it is born, is not in any relationships with other objects. That is, the roles in the new node are “disconnected”. It is the **read_graph()** operation’s job to correctly establish new relationships.

Figure 8-6 Internalizing a node returns the new object and the corresponding roles.

If an appropriate factory to internalize the roles cannot be found, the **NoFactory** exception is raised. The exception value indicates the key used to find the factory.

In addition to the **NoFactory** exception, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the internalized node, **NO_RESOURCES** will be raised.

8.6.6 The Role Interface

The **Role** interface defines operations to externalize and internalize a role. The **Role** interface also defines an operation to return the propagation value for the externalize operation.

The implementation of a **CosStream::Node** operation can call these operations on roles. For example, an implementation of **externalize** on a node can call the **externalize** operation on the **Role**.

8.6.6.1 Externalizing a Role

void externalize_role (in CosStream::StreamIO sio);

The **externalize_role()** operation transfers the role’s state to the stream **sio**.

8.6.6.2 Internalizing a Role

void internalize_role (in CosStream::StreamIO sio);

The **internalize_role()** operation causes a role to read its state from the stream given by **sio**.

8.6.6.3 Getting a Propagation Value

```
CosGraphs::PropagationValue externalize_propagation (
    in RelationshipHandle rel,
    in CosRelationships::RoleName toRoleName,
    out boolean sameForAll);
```

The **externalize_propagation()** operation returns the propagation value to the role **toRoleName** for the externalization operation and the relationship **rel**. If the role can guarantee that the propagation value is the same for all relationships in which it participates, **sameForAll** is true.

8.6.7 The Relationship Interface

The **Relationship** interface defines operations to externalize and internalize a relationship. The **Relationship** interface also defines an operation to return the propagation values for the exteranlize operations.

8.6.7.1 Externalizing the Relationship

```
void externalize_relationship (
    in CosStream::StreamIO sio);
```

The **externalize_relationship()** operation transfers the role's state to the stream **sio**.

8.6.7.2 Internalizing the Relationship

```
void internalize_relationship(
    in CosStream::StreamIO sio,
    in CosGraphs::NamedRoles newRoles);
```

The **internalize_relationship()** operation internalizes the state of a relationship from the stream given by **sio**.

The values of the internalized relationship's attributes are defined by the implementation of this operation. However, the **named_roles** attribute of the newly created relationship must match **newRoles**. That is, the internalized relationship relates objects represented by **newRoles** parameter, not the by the original relationship's named roles.

8.6.7.3 Getting a Propagation Value

```
CosGraphs::PropagationValue externalize_propagation (
    in CosRelationships::RoleName fromRoleName,
    in CosRelationships::RoleName toRoleName,
    out boolean sameForAll);
```

The **propagation_for()** operation returns the relationship's propagation value from the role **fromRoleName** to the role **toRoleName** for the externalization operation. If the role named by **fromRoleName** can guarantee that the propagation value is the same for all relationships in which it participates, **sameForAll** is true.

8.6.8 The PropagationCriteriaFactory Interface

The **CosGraphs** module in the Relationship Service defines a general service for traversing a graph of related objects. The service accepts a "call-back" object supporting the **CosGraphs::TraversalCriteria** interface. Given a node, this object defines which edges to emit and which nodes to visit next.

The **PropagationCriteriaFactory** creates a **TraversalCriteria** object that determines which edges to emit and which nodes to visit based on propagation values for the compound externalization operations.

8.6.8.1 Create a Traversal Criteria Based on Externalization Propagation

CosGraphs::TraversalCriteria create_for_externalize( );

The **create_for_externalize** operation returns a **TraversalCriteria** object for an operation op that determines which edges to emit and which nodes to visit based on propagation values for op. For a more detailed discussion see the Relationship Service chapter.

8.7 Specific Externalization Relationships

The Relationship Service defines two important relationships: containment and reference. Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment is represented by a relationship with two roles: the **ContainsRole**, and the **ContainedInRole,** Similarly, reference is represented by a relationship with two roles: **ReferencesRole** and **ReferencedByRole**.

Compound externalization adds externalization semantics to these specific relationships. That is, it defines propagation values for containment and reference.

8.8 The CosExternalizationContainment Module

The **CosExternalizationContainment** module defines the following interfaces:

- **Relationship** interface
- **ContainsRole** interface
- **ContainedInRole** interface

```
//File: CosExternalizationContainment.idl
//Part of the Externalization Service
// modified from version 1.0 to use CosStream module
//     instead of CosCompoundExternalization

#ifndef _COS_EXTERNALIZATION_CONTAINMENT_IDL_
#define _COS_EXTERNALIZATION_CONTAINMENT_IDL_

#include <CosContainment.idl>
#include <CosStream.idl>

#pragma prefix "omg.org"

module CosExternalizationContainment {

    interface Relationship :
        CosStream::Relationship,
        CosContainment::Relationship {};

    interface ContainsRole :
        CosStream::Role,
        CosContainment::ContainsRole {};

    interface ContainedInRole :
        CosStream::Role,
        CosContainment::ContainedInRole {};
};
#endif /* ifndef _COS_EXTERNALIZATION_CONTAINMENT_IDL_*/
```

The **CosExternalizationContainment** module does not define new operations. It merely “mixes in” interfaces from the **CosStream** and **CosContainment** modules. Although it does not add any new operations, it refines the semantics of these operations:

The **CosExternalizationContainment::ContainsRole::propagation_for** operation returns the following:

operation	ContainsRole to ContainedInRole
externalize	deep

The **CosExternalizationContainment::ContainedInRole::propagation_for()** operation returns the following::

operation	ContainedInRole to ContainsRole
externalize	none

The **CosRelationships::RoleFactory::create_role()** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosStream::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationContainment::ContainsRole** and **CosExternalizationContainment::ContainedInRole**. It will raise **MaxCardinalityExceeded** if the **CosExternalizationContainment::ContainedInRole** is already participating in a relationship.

8.9 The CosExternalizationReference Module

The **CosExternalizationReference** module defines these interfaces:

- **Relationship** interface
- **ReferencesRole** interface
- **ReferencedByRole** interface

```
//File: CosExternalizationReference.idl
//Part of the Externalization Service
// modified from version 1.0 to use CosStream module
//     instead of CosCompoundExternalization

#ifndef _COS_EXTERNALIZATION_REFERENCE_IDL_
#define _COS_EXTERNALIZATION_REFERENCE_IDL_

#include <CosReference.idl>
#include <CosStream.idl>

#pragma prefix "omg.org"

module CosExternalizationReference {

    interface Relationship :
        CosStream::Relationship,
        CosReference::Relationship {};

    interface ReferencesRole :
        CosStream::Role,
        CosReference::ReferencesRole {};
```

```
    interface ReferencedByRole :
        CosStream::Role,
        CosReference::ReferencedByRole {};
};
#endif /* ifndef _COS_EXTERNALIZATION_REFERENCE_IDL_ */
```

The **CosExternalizationReference** module does not define new operations. It merely "mixes in" interfaces from the **CosStream** and **CosReference** modules. Although it does not add any new operations, it refines the semantics of these operations:

The **CosExternalizationReference::ReferencesRole::propagation_for()** operation returns the following:

operation	ReferencesRole to ReferencedByRole
externalize	none

The **CosExternalizationReference::ReferencedByRole::propagation_for()** operation returns the following::

operation	ReferencedByRole to ReferencesRole
externalize	none

The **CosRelationships::RoleFactory::create_role()** operation will raise the **RelatedObjectTypeError** if the related object passed as a parameter does not support the **CosStream::Node** interface.

The **CosRelationships::RelationshipFactory::create()** operation will raise **DegreeError** if the number of roles passed as arguments is not 2. It will raise **RoleTypeError** if the roles are not **CosExternalizationReference::ReferencesRole** and **CosExternalizationReference::ReferencedByRole**.

8.10 Standard Stream Data Format

An externalization client may create a stream that supports a specific external representation data format that is intended to be portable across different CORBA implementations and on different CPU hardware. A client creates such a **Stream** object using a factory found by specifying a Key whose only **NameComponent** has an **NameComponent**::id whose value is the string literal "StandardExternalizationFormat".

That format is described in this section.

8.10.1 OMG Externalized Object Data

1 byte

tag byte = x'F0'	Key info	Object info

A leading "tag" byte with a value of x"F0" marks the beginning of an object's externalized data. Following this is data associated with a Key that can be used to internalize the object. The key information is then followed by the data written to the **StreamIO** for the object's state.

Key Info

1 byte

length = i	1st id string	2nd id string	. . .	i'th id string

The key information consists of a byte containing an integer value, "i", that indicates how many **CosNaming::NameComponent**s make up the associated Key.

This byte is followed by "i" null-terminated sequences of char values that represent the **CosNaming::NameComponent::id** values for the Key. These values correspond to the C mapping of a CORBA string type. The **NameComponent::kind** values are not stored in this external data format.

Object Info

1 byte   1 byte

tag byte	data value	tag byte	data value	. . .

The object information is the sequence of bytes generated for one or more **write_<type>** operation. For each **write_<type>** operation, a single "tag" byte identifying the type of the primitive data is followed by the data. The tag byte gives the internalization implementation enough information to skip past object state for objects that cannot be created, for example when a compatible implementation cannot be found on the internalizing ORB.

The tag byte values, and data formats for each type are as indicated below for basic CORBA data types:

Table 8-4 CORBA Tag Byte Values and Data Formats

tag	CORBA type	data format
x'F1'	Char	one byte
x'F2'	Octet	one byte

Table 8-4 CORBA Tag Byte Values and Data Formats

x'F3'	Unsigned Long	four bytes, big-endian format
x'F4'	Unsigned Short	two bytes, big-endian format
x'F5'	Long	four bytes, big-endian format
x'F6'	Short	two bytes, big-endian format
x'F7'	Float	four bytes, IEEE 754 single precision format, sign bit in first byte
x'F8'	Double	eight bytes, IEEE 754 double precision format, sign bit first byte
x'F9'	Boolean	TRUE=>one byte==1, FALSE=>one byte==0
x'FA'	String	null-terminated sequence of bytes

8.10.2 Externalized Repeated Reference Data

1	4	(bytes)
x'04'	Object number	

This format is used only when multiple objects reference the same object. Instead of storing the referenced object multiple times, the duplicate reference objects are stored in this format. Note that the object is represented by a long object number which indicates that the object has been stored already.

8.10.3 Externalized NIL Data

1 (byte)
x'05'

This is a special format used to indicate that there is no object stored in the stream.

8.11 References

1. James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations." *OOPSLA 1988 Proceedings*, pg. 285-296

2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, “Object-oriented Modeling and Design.” Prentice Hall, 1991.

3. Grady Booch, “Object Oriented Design with Applications.” The Benjamin/Cummings Publishing Componay, Inc., 1991.

Relationship Service Specification 9

9.1 Service Description

Distributed objects are frequently used to model entities in the real world. As such, distributed objects do not exist in isolation. They are related to other objects.

Consider some examples of real world entities and relationships:

- A person ***owns*** cars; a car is ***owned by*** one or more persons.
- A company ***employs*** one or more persons; a person is ***employed by*** one or more companies.
- A document ***contains*** figures; a figure is ***contained in*** a document.
- A document ***references*** a book; a book is ***referenced by*** one or more documents.
- A person ***checks out*** books from libraries. A library ***checks out*** books to people. A book is ***checked out*** by a person from a library.

These examples demonstrate several relationships:

- Ownership relationships between people and cars
- Employment relationships between companies and people
- Containment relationships between documents and figures
- Reference relationships between books and documents
- Check out relationships between people, books and libraries.

Such relationships can be characterized along a number of dimensions:

Type

Related entities and the relationships themselves are typed. In the examples, ***employment*** is an relationship defined between ***people*** and ***companies***. The type of the relationship constrains the types of entities in the relationship; a company cannot employ a monkey since a monkey is not a person. Furthermore, employment is distinct from other relationships between people and companies.

The roles of entities in relationships

A relationship is defined by a set of roles that entities have. In an employment relationship, a company plays an ***employer*** role and a person plays an ***employee*** role.

A single entity can have different roles in distinct relationships. Notice that a person can play the owner role in an ownership relationship and the employee role in an employment relationship.

Degree

Degree refers to the number of required roles in a relationship. The check out relationship is a ternary relationship; it has three roles: the borrower role, the lender role and the material role. A person plays the borrower role, a library plays the lender role and a book plays the material role. Ownership, employment, containment and reference, on the other hand, are of degree 2, or binary relationships.

Cardinality

For each role in a relationship type, the maximum cardinality specifies the maximum number of relationships that may involve that role.

The containment relationship is a many-to-one relationship; a document contains many figures; a figure is contained in exactly one document. A many-to-many relationship is between two sets of entities. The ownership example is a many-to-many relationship; a person can own multiple cars; a car can have multiple owners. The check out relationship is a many-to-one-to-many relationship. A person can check out many books from many libraries. A book is checked out by one person from one library and a library can loan many books to many people.

Relationship Semantics

Relationships often have relationship-specific semantics; that is they define operations and attributes. For example, ***job title*** is an attribute of the employment relationship, while it is not an attribute of an ownership relationship. Similarly, ***due date*** is an attribute of the check out relationship.

For more discussion on object-oriented modeling and design with relationships, see [2.].

9.1.1 Key Features of the Relationship Service

- The Relationship Service allows entities and relationships to be explicitly represented. Entities are represented as CORBA objects. The service defines two new kinds of objects: *relationships* and *roles*. A role *represents* a CORBA object in an relationship. A relationship is created by passing a set of roles to a relationship factory.
- Relationships of arbitrary degree can be defined.
- Type and cardinality constraints can be expressed and checked. Exceptions are raised when cardinality and type constraints are violated. The Relationship Service does not define a new type system. Instead, the IDL type system is used to represent relationship and role types. This allows the service to leverage CORBA solutions for type federation.

- The *Relationship* interface can be extended to add relationship specific attributes and operations. Similarly, the *Role* interface can be extended to add role specific attributes and operations.
- The Relationship Service defines three levels of service: base, graph, and specific.
- The base level defines relationships and roles.
- When objects are related, they form graphs of related objects. The graph level extends the base level service with nodes and traversal objects. Traversal objects iterate through the edges of a graph. Traversals are useful in implementing compound operations on graphs, among other things.
- Specific relationships are defined by the third level.

4. A conforming Relationship Service implementation must implement level 1 or levels 1 and 2 or levels 1, 2 and 3.
   - The Relationship Service requires a notion of object identify. As such, it defines a simple, efficient mechanism for supporting object identity in a heterogeneous, CORBA-based environment. We believe the mechanism to be of general utility for other services.
   - Distributed implementations of the Relationship Service can have navigation performance and availability similar to CORBA object references; role objects can be collocated with their objects and need not depend on a centralized repository of relationship information. As such, navigating a relationship can be a local operation.
   - The Relationship Service allows so-called immutable objects to be related. There are no required interfaces that objects being related must support. As such, objects whose state and implementation were defined prior to the definition of the Relationship Service can be related objects.
   - The Relationship Service allows graphs of related objects to be traversed without activating related objects.
   - The Relationship Service is extensible. Programmers can define additional relationships.

9.1.2 The Relationship Service vs. CORBA Object References

CORBA: Common Object Request Broker Architecture and Specification defines object references that clients use to issue requests on objects. Object references can be stored persistently. When is it appropriate to use object references and when is it appropriate to use the Relationship Service?

The Relationship Service is appropriate to use when an application needs any of the following capabilities that are not available with CORBA object references:

Relationships that Are Multidirectional

When objects are related using the Relationship Service, the relationship can be navigated from any role to any other role. The service maintains the relationship between related objects. CORBA object references, on the other hand, are unidirectional. Objects that posses CORBA object references to each other can only do so in an ad hoc fashion; there is no way to maintain and manipulate the relationship between the objects.

Relationships that Allow Third Party Manipulation

Since roles and relationships are themselves CORBA objects, they can be exported to third parties. This allows third parties to manipulate the relationship. For example a third party could create, destroy or navigate the relationship. Third parties cannot manipulate object references.

Traversals that Are Supported for Graphs of Related Objects

When objects are related using the Relationship Service, they form graphs of related objects. Interfaces are defined by the Relationship Service to support traversing the graph.

Relationships and Roles that Can Be Extended with Attributes and Behavior

Relationships have relationship-specific semantics. For example, the employment relationship has a job title attribute. Since relationships and roles are objects with well-defined OMG IDL interfaces, they can be extended through OMG IDL inheritance to add such relationship-specific attributes and operations.

9.1.3 Resolution of Technical Issues

Modeling and Relationship Semantics

An application designer models a problem as a set of objects and the relationships between those objects. Using OMG IDL, the application designer directly represents the objects of the model. Using the Relationship Service, the application designer directly represents the roles and relationships of the model.

The *Relationship* and *Role* interfaces can be extended using OMG IDL inheritance to add relationship and role specific attributes and operations. For example, a designer might define the employment relationship to have an operation returning a job title.

Managing Relationships

The *RelationshipFactory* interface defines an operation to create a relationship, given a set of roles. The *Role* and *Relationship* interfaces define operations to delete and navigate relationships between objects.

Constraining Relationships

Type, cardinality and degree constraints on relationships are expressed in the interfaces.

The RoleFactory::create_role operation can raise a RelatedObjectTypeError exception. This allows implementations of the *Role* interface to place further constraints on the type of the related objects. For example, an *EmployedByRole* can ensure related objects are people. An attempt to have it represent a monkey would raise a RelatedObjectTypeError exception.

Similarly, the RelationshipFactory::create operation can raise a RoleTypeError exception. This allows implementations of the *Relationship* interface to put constraints on the type of the roles. For example an Employment relationship can ensure there is an *EmployerRole* and an *EmployeeRole*.

The RelationshipFactory::create operation can also raise a DegreeError exception. This ensures that there are the correct number of roles.

Maximum cardinality constraints are enforced by the role objects themselves. A role can raise a MaxCardinalityExceeded exception and refuse to participate in a relationship if its maximum cardinality would be exceeded. Roles define an operation to ask if their minimum cardinality constraint is being met.

Referential Integrity

If the Relationship Service is used in an environment supporting transactions, strict referential integrity is achieved. That is, if an related object refers to another (via a relationship), then the other related object will also refer to it. Without transactions, strict referential integrity cannot be achieved since a failure during execution of the relationship construction protocol could cause a dangling reference.

Relationships and Roles as First Class Objects

Our design defines both relationships and roles as first class objects. This is extremely important because it encapsulates and abstracts the state to represent the relationship, allows third party manipulation of the relationship and allows the roles and relationships themselves to support operations and attributes.

Different Models for Navigating and Constructing Relationships

The Relationship Service defines interfaces for constructing and navigating relationships component-by-component. These building block operations can be used by a higher-level service, such as a query service.

Efficiency Considerations

Our design has several features that allow for highly optimized implementations. Performance optimizations are achieved by clustering and/or *c*aching of connection information.

Clients can cluster related objects and their roles by their selection of factories.

Our design defines the containment relationship logically. It does not imply physical clustering of state or execution, However, it serves as a good hint to implementations for clustering. An environment can choose to cluster containers and contained objects.

The get_other_related_object operation can be implemented to *cache* remote related objects. The cached information is immutable; once a relationship is established, the roles and related objects will not change.

9.2 Service Structure

This section provides information about the levels of service; the specification is organized around these levels. It also describes the hierarchy of Relationship Service interfaces and explains the main purpose of each interface.

9.2.1 Levels of Service

The Relationship Service defines three levels of service: base relationships, graphs of related objects, and specific relationships. The specification is organized around these levels.

Level One: Base Relationships

The *Relationship* and *Role* interfaces define the base Relationship Service. Figure 9-1 illustrates two instances of the containment relationship. The document plays the container role; the figure and the logo play the containee role.

The diamond is an object supporting the *Relationship* interface. The small circles are objects supporting the *Role* interface.

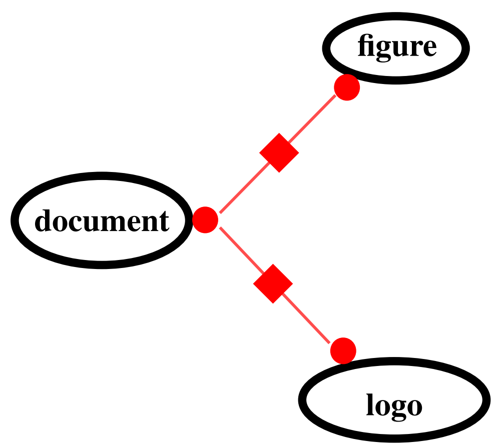


Figure 9-1 Base relationships.

Roles represent objects in relationships. Roles have a maximum cardinality. As illustrated, the container role can be involved in many instances of a relationship. The containee roles can only be involved in a single instance of a relationship.

Figure 9-2 illustrates the navigation functionality of relationships; for example the arrow between a role and another role indicates it is possible to navigate from one role to another. The arrow does not, however, indicate that the object reference to the other role is necessarily stored by the role.

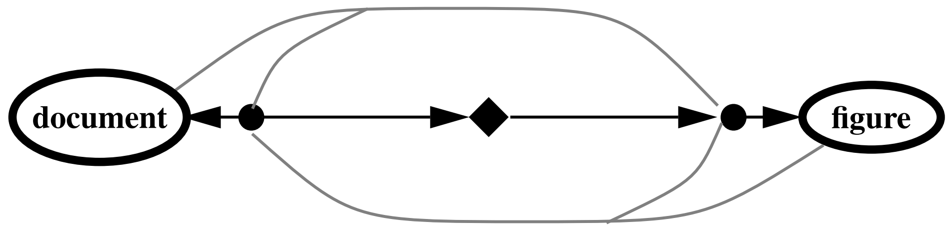


Figure 9-2 Navigation functionality of base relationships

Table 9-1 lists the interfaces to support relationships and roles.

Level Two: Graphs of Related Objects

Distributed objects do not exist in isolation. They are connected together. Objects connected together form graphs of related objects. The Relationship Service defines the *Traversal* interface. The *Traversal* interface defines an operation to traverse a graph. The traversal object cooperates with extended roles supporting the *CosGraphs::Role* interface and objects supporting the *Node* interface.

Figure 9-3 illustrates a graph of related objects. The folder, the figure, the logo and the book all support the *Node* interface. The small circles are roles supporting the *CosGraphs::Role* interface.

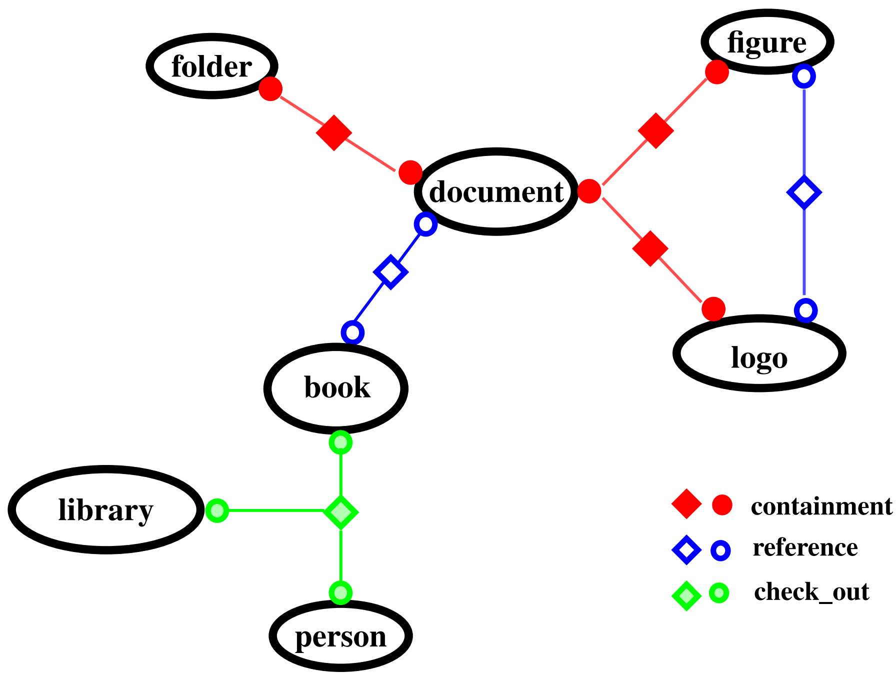


Figure 9-3 An example graph of related objects.

Table 9-3 lists the interfaces to support graphs of related objects.

Level Three: Specific Relationships

Containment and reference are two important relationships. The Relationship Service defines these two binary relationships. Table 9-4 and Table 9-5 list the interfaces defining specific relationships.

9.2.2 Hierarchy of Relationship Interface

The relationship interfaces are arranged into the interface hierarchy illustrated in Figure 9-4.

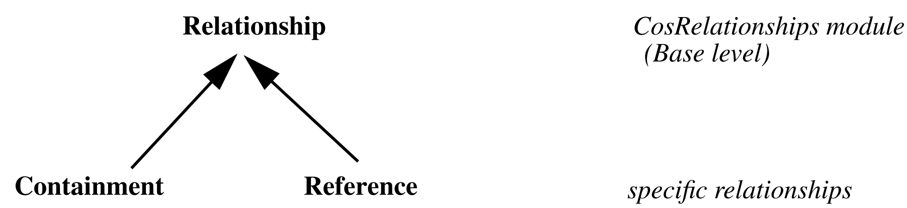


Figure 9-4 Relationship interface hierarchy

9.2.3 Hierarchy of Role Interface

The role interfaces are arranged into the interface hierarchy illustrated in Figure 9-5.

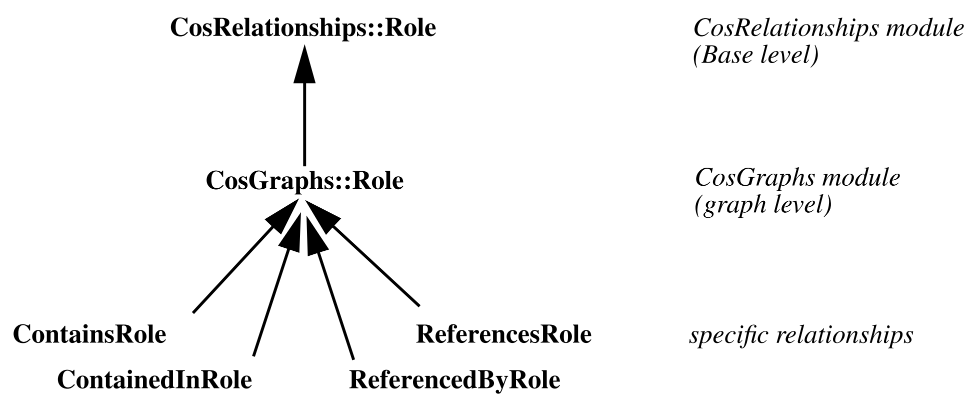


Figure 9-5 Role interface hierarchy

The *Role* interface defines operations to efficiently navigate relationships between related objects.

The *CosGraphs::Role* interface defines an operation to return the edges that involve the role. This is used by the traversal service defined at the graph level.

Finally, *ContainsRole*, *ContainedInRole*, *ReferencesRole* and *ReferencedByRole* are specific roles for two important relationships: containment and reference.

9.2.4 Interface Summary

The Relationship Service defines interfaces to support the functionality described in section 9.2.

Table 9-1 through Table 9-5 give high level descriptions of the Relationship Service interfaces.

Table 9-1 Interfaces defined in the *CosObjectIdentity* module

Interface	Purpose	IPrimary Clients
CosObjectIdentity::		
IdentifiableObject	To determine if two objects are identical.	There are many clients. The graph level of the Relationship Service is one.

Table 9-2 Interfaces defined in the *CosRelationships* module

Interface	Purpose	Primary Clients
CosRelationships::		
Relationship	Represents an instance of a relationship type.	Clients that navigate between related objects.
RelationshipFactory	Supports the creation of relationships.	Clients establishing relationships.
Role	Defines navigation operations for relationships. Implements type and cardinality constraints.	Clients that navigate between related objects. Relationship factories.
RoleFactory	Supports the creation of roles.	Objects participating in relationships.
RelationshipIterator	Iterates the relationships in which a particular role object participates.	Clients that navigate relationships.

Table 9-3 Interfaces defined in the *CosGraphs* module

Interface	Purpose	Primary Client(s)
CosGraphs::		
Traversal	Defines an operation to traverse a graph, given a starting node and traversal criteria.	Clients that want a standard service to traverse graphs.
TraversalFactory	Supports the creation of a traversal object.	Clients that want a standard service to traverse graphs.
TraversalCriteria	Provides navigation behavior between nodes.	Traversal implementations.
Role	Extends the CosRelationships::Role interface to return edges	Clients that traverse graphs of related objects.
EdgeIterator	Returns additional edges from a role.	Clients that traverse graphs of related objects.
Node	Defines operations for a related object to reveal its roles.	Clients that traverse graphs of related objects.
NodeFactory	Supports the creation of nodes.	Clients that create nodes in graphs.

Table 9-4 Interfaces defined in the *CosContainment* module

Interface	Purpose	Primary Client(s)
CosContainment::		
Relationship	one-to-many relationship	Clients that depend on Containment relationship type.
ContainsRole	Represents an object that contains other objects.	Clients that navigate containment relationships between objects.
ContainedInRole	Represents an object that is contained in other objects.	Clients that navigate containment relationships between objects.

Table 9-5 Interfaces defined in the *CosReference* module

Interface	Purpose	Primary Clients
CosReference::		
Relationship	many-to-many relationship	Clients that depend on the reference relationship type.
ReferencesRole	Represents an object that references other objects.	Clients that navigate reference relationships between objects.
ReferencedByRole	Represents an object that is referenced by other objects.	Clients that navigate reference relationships between objects.

9.3 The Base Relationship Model

The base level of the Relationship Service defines interfaces that support relationships between two or more CORBA objects. Objects that participate in a relationship are called related objects. Relationships that share the same semantics form *relationship types*. A relationship is an instance of a relationship type and has an identity.

Each related object is connected with the relationship via a role. Roles are objects which characterize a related object‘s participation in a relationship type. Role types are used for expressing the role´s characteristics by an IDL interface. Cardinality represents the number of relationship instances connected to a role. Degree represents the number of roles in a relationship. All characteristics are expressed by corresponding IDL interfaces. Relationship and role types are built by subtyping the Relationship and Role interfaces.

Figure 9-6 gives a graphical representation of a simple relationship type. It illustrates that documents reference books. Documents are in the *ReferencesRole* and books are in the *ReferencedByRole*. Documents, reference, the roles and books are all types; there are interfaces (written in OMG IDL) for all five.

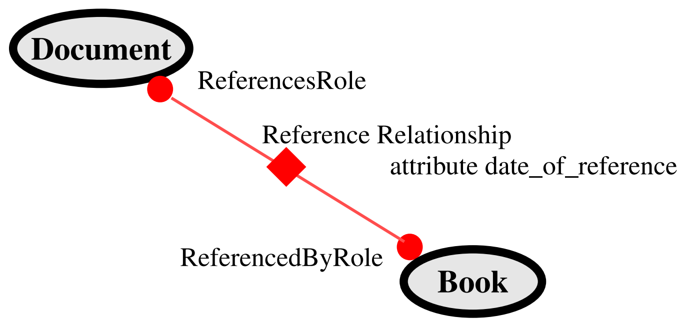


Figure 9-6 Simple relationship type: documents reference books

Figure 9-7, on the other hand, gives a graphical representation of an instance of a relationship type. It illustrates that “my document”, an instance of Document, references “War and Peace”, an instance of Book. Most of the figures in this specification represent instances of related objects, roles and relationships. Figures describing object and relationship type are clearly marked.

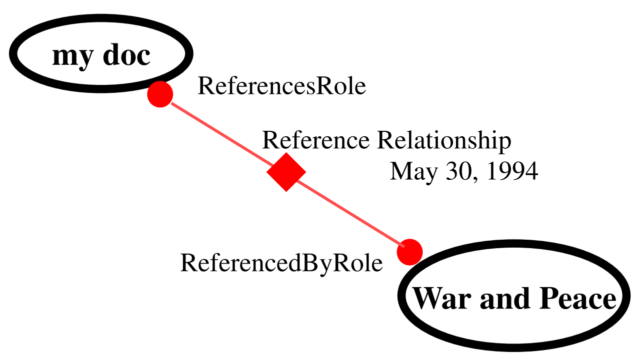


Figure 9-7 Simple relationship instance: my document references the book “War and Peace“

9.3.1 Relationship Attributes and Operations

Relationships may have attributes and operations. For example, the reference relationship of Figure 9-6 has an attribute indicating the date the reference from the document to the book was established.

Rationale

If relationships are not allowed to define attributes and operations, they will have to be assigned to one of the related objects. This approach is prone to misunderstandings and inconsistencies. The approach to define an artificial related object, which then carries the attributes, is equally unsatisfactory.

The date attribute of the example of Figure 9-7 is clearly an attribute of the relationship, not one of related objects. It cannot be an attribute of “my document” since “my document” can reference many books on different dates. Similarly, it cannot be an attribute of “War and Peace” since “War and Peace” can be referenced by many books on different dates.

9.3.2 Higher Degree Relationships

The Reference relationship in Figure 9-6 is a *binary* relationship; that is, it is defined by two roles. The Relationship Service can also support relationships with more than two roles. The fact that three or more related objects may be part of a relationship can be expressed directly by means of the same concept as in the binary case. The *degree* represents the number of roles in a relationship. The Relationship Service supports higher degree relationships, that is relationships with degree greater than two.

Figure 9-8 shows a ternary “check out” relationship between books, libraries and persons. The semantics of this relationship is that a person borrows a book from a library. The relationship also defines an attribute that indicates the date when the book is due to be returned by the person to the library.

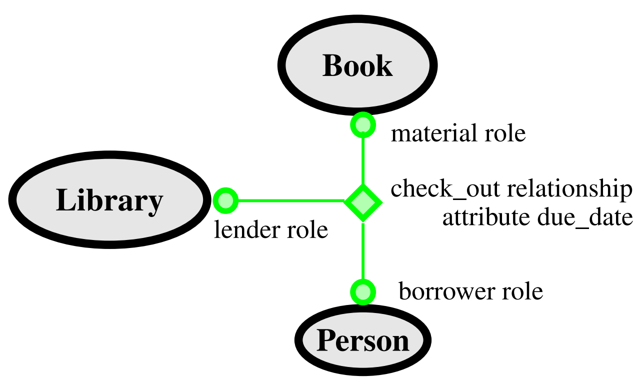


Figure 9-8 A ternary check-out relationship type between books, libraries and persons.

Rationale

The Relationship Service represents higher degree relationships directly. It clearly defines the number of expected related objects as well as other integrity constraints. It is more readable, more understandable and easier to enforce consistency constraints for related objects with a direct representation than with alternative representations that simulate higher degree relationships using a set of binary relationships. When simulating higher degree relationships, the relationship information is spread over multiple object and relationship type definitions, as are the corresponding integrity constraints.

Figure 9-9 shows an alternative representation of the ternary relationship from Figure 9-8 using binary relationships. Note that the first representation is not equivalent to that of Figure 9-8 since cardinalities and other integrity constraints cannot be expressed correctly in this alternative representation.

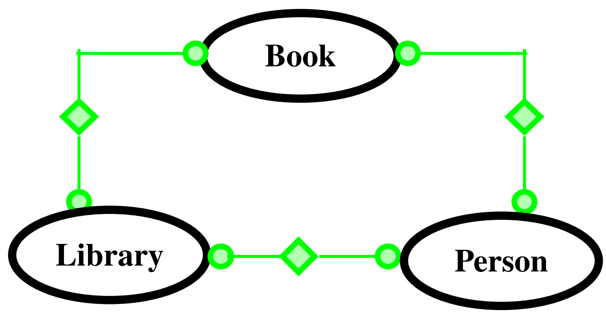


Figure 9-9 An unsatisfactory representation of the ternary check-out relationship using binary relationships.

Figure 9-10 illustrates a second alternative representation of the ternary relationship of Figure 9-8. It uses an additional (artificial) related object type. This representation is equivalent to Figure 9-8 if *Check-out* is constrained to participate in exactly one instance of each of the three binary relationship types. However, this alternative needs three relationship types and one additional related object type *(Check-out)* instead of only one relationship type, and therefore is much more complex and harder to capture when compared to the representation using one relationship type with degree 3.

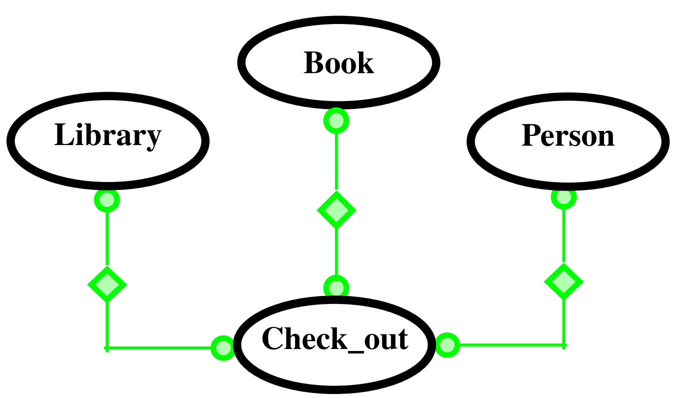


Figure 9-10 Another unsatisfactory representation

Since the Relationship Service supports higher order relationships directly, the user of the service need not resort to the unsatisfactory representations using binary relationships of Figure 9-9 and Figure 9-10.

9.3.3 Operations

The base level of the Relationship Service provides operations to:

- Create role and relationship objects
- Navigate relationships
- Destroy roles and relationships
- Iterate over the relationships in which a role participates

Creation

Roles are constructed independently using a role factory. Roles represent an existing related object that is passed as a parameter to the RoleFactory::create operation. When creating a new role object, the type of the related object can be checked by the factory. The minimum and maximum cardinality, e.g. the minimal and the maximal number of relationship instances to which the new role object may be connected, are indicated by attributes on the factory.

Figure 9-11 illustrates a newly created role.

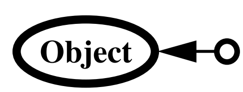


Figure 9-11 Creating a role for an object

A new relationship is created by passing a sequence of named roles to a factory for the relationship. The expected degree and role types for the new relationship are indicated by attributes on the factory. During the creation of the new relationship, the role types and the maximum cardinality can be checked. Duplicate role names are not allowed since the names are used to distinguish the roles in the scope of the relationship.

When creating a relationship, the factory creates “links” between the roles and the relationship using the link operation on the role.

Figure 9-12 illustrates a fully established binary relationship. Figure 9-12 represents *navigation functionality*; it does not necessarily represent stored object references. A variety of implementation strategies are described in section 9.3.5.

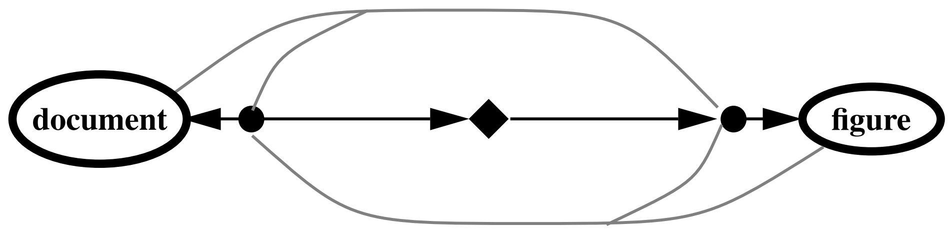


Figure 9-12 A fully established binary relationship

Navigation

Figure 9-12 illustrates the navigational functionality of a relationship. In particular,

- a relationship defines an attribute that indicates a read-only attribute that indicates the named roles of the relationship,
- a role defines a read-only attribute that indicates the related object that the role represents,
  - A role supports the get_other_role operation, that given a relationship object and a role name, returns the other role object,
  - A role supports the get_other_related_object operation, that given a relationship object and a role name, returns the related object that the named role represents in the relationship and
  - A role supports the get_relationships operation which returns the relationships in which the role participates.

Destruction

For both roles and relationship objects, the Relationship Services introduces a destroy operation. The destroy operation for relationship objects also destroys the links between the relationship and all of the role objects.

9.3.4 Consistency Constraints

For each role two cardinalities are defined: minimum and maximum.

- The minimum cardinality indicates the minimum number of relationship instances in which a role must participate.
- The maximum cardinality indicates the maximum number of relationship instances in which a role can participate.

Maximum cardinality constraint can be checked when relationships are created. Note that the relationship mechanism cannot, by itself, enforce the minimum cardinality constraint. However, a role can be asked explicitly if it meets its minimum cardinality constraint using the check_minimum_cardinality operation.

Type integrity is preserved by CORBA mechanisms because related objects, roles and relationships are instances of CORBA object types. Type constraints can be checked when roles and relationships are created.

9.3.5 Implementation Strategies

Figure 9-12 illustrates the navigational functionality of a fully established binary relationship. There are a variety of implementation strategies possible. The get_other_role and the get_other_related_object operations can be:

- Implemented by caching object references to other roles and related objects, or
- Computed when needed using the relationship object.

The appropriate implementation strategy typically depends on distribution boundaries. If the roles and relationship objects are clustered, then only storing the values at the relationship object optimizes space. If, on the other hand, the roles and the related objects are clustered, caching object references to other roles and related objects at the roles allows the relationship to be efficiently navigated without involving a remote relationship object.

Role implementations that cache object references to other roles and related objects need not worry about updating the cache. Once the related objects and relationships are established, they cannot be changed.

9.3.6 The CosObjectIdentity Module

CORBA: Common Object Request Broker Architecture and Specification does not define a notion of object identity for objects. The Relationship Service requires object identity for the objects it defines. As such, the Relationship Service assumes the CosObjectIdentity module specified in Figure 9-13 . This is defined in a separate module; other Object Services may find this module to be generally useful.

```
module CosObjectIdentity {

	typedef unsigned long ObjectIdentifier;

	interface IdentifiableObject {
		readonly attribute ObjectIdentifier constant_random_id;
		boolean is_identical (
			in IdentifiableObject other_object);
	};

};
```

Figure 9-13 The CosObjectIdentity Module

The IdentifiableObject Interface

Objects that support the *IdentifiableObject* interface implement an attribute of type *ObjectIdentifier* and the is_identical operation. This mechanism provides an efficient and convenient method of supporting object identity in a heterogeneous CORBA-based environment.

constant_random_id

```
readonly attribute ObjectIdentifier constant_random_id;
```

Objects supporting the *IdentifiableObject* interface define an attribute of type *ObjectIdentifier*. The value of the attribute must not change during the lifetime of the object.

A typical client use of this attribute is as a key in a hash table. As such, the more randomly distributed the values are, the better.

The value of this attribute is not guaranteed to be unique; that is, another identifiable object can return the same value. However, if objects return different identifiers, clients can determine that two identifiable objects are ***not*** identical.

To determine if two identifiable objects ***are*** identical, the is_identical operation must be used.

is_identical

```
boolean is_identical (
in IdentifiableObject other_object);
```

The is_identical operation returns *true* if the object and the other_object are identical. Otherwise, the operation returns *false*.

9.3.7 The CosRelationships Module

The CosRelationships module defines the interfaces of the base level Relationship Service. In particular, it defines

- *Relationship* and *Role* interfaces to represent relationships and roles,
- *RelationshipFactory* and *RoleFactory* interfaces to create relationships and roles
- *RelationshipIterator* interface to enumerate the relationships in which a role participates

The CosRelationships module is shown in Figure 9-14.

```
#include <ObjectIdentity.idl>

module CosRelationships {

    interface RoleFactory;
    interface RelationshipFactory;
    interface Relationship;
    interface Role;
    interface RelationshipIterator;

    typedef Object RelatedObject;
    typedef sequence<Role> Roles;
    typedef string RoleName;
    typedef sequence<RoleName> RoleNames;

    struct NamedRole {RoleName name; Role aRole;};
    typedef sequence<NamedRole> NamedRoles;

    struct RelationshipHandle {
        Relationship the_relationship;
        CosObjectIdentity::ObjectIdentifier constant_random_id;
    };
    typedef sequence<RelationshipHandle> RelationshipHandles;

    interface RelationshipFactory {
        struct NamedRoleType {
            RoleName name;
            ::CORBA::InterfaceDef named_role_type;
        };
        typedef sequence<NamedRoleType> NamedRoleTypes;
        readonly attribute ::CORBA::InterfaceDef relationship_type;
        readonly attribute unsigned short degree;
        readonly attribute NamedRoleTypes named_role_types;
        exception RoleTypeError {NamedRoles culprits;};
        exception MaxCardinalityExceeded {
            NamedRoles culprits;};
        exception DegreeError {unsigned short required_degree;};
        exception DuplicateRoleName {NamedRoles culprits;};
        exception UnknownRoleName {NamedRoles culprits;};

        Relationship create (in NamedRoles named_roles)
            raises (RoleTypeError,
                MaxCardinalityExceeded,
                DegreeError,
                DuplicateRoleName,
                UnknownRoleName);
    };
```

Figure 9-14 The CosRelationships Module

```
interface Relationship :
        CosObjectIdentity::IdentifiableObject {
    exception CannotUnlink {
        Roles offending_roles;
    };
    readonly attribute NamedRoles named_roles;
    void destroy () raises(CannotUnlink);
};

interface Role {
    exception UnknownRoleName {};
    exception UnknownRelationship {};
    exception RelationshipTypeError {};
    exception CannotDestroyRelationship {
        RelationshipHandles offenders;
    };
    exception ParticipatingInRelationship {
        RelationshipHandles the_relationships;
    };
        readonly attribute RelatedObject related_object;
    RelatedObject get_other_related_object (
            in RelationshipHandle rel,
            in RoleName target_name)
        raises (UnknownRoleName,
            UnknownRelationship);
    Role get_other_role (in RelationshipHandle rel,
            in RoleName target_name)
        raises (UnknownRoleName, UnknownRelationship);
    void get_relationships (
            in unsigned long how_many,
            out RelationshipHandles rels,
            out RelationshipIterator iterator);
    void destroy_relationships()
        raises(CannotDestroyRelationship);
    void destroy() raises(ParticipatingInRelationship);
    boolean check_minimum_cardinality ();
    void link (in RelationshipHandle rel,
            in NamedRoles named_roles)
        raises(RelationshipFactory::MaxCardinalityExceeded,
            RelationshipTypeError);
    void unlink (in RelationshipHandle rel)
        raises (UnknownRelationship);
};

interface RoleFactory {
    exception NilRelatedObject {};
    exception RelatedObjectTypeError {};
    readonly attribute ::CORBA::InterfaceDef role_type;
```

Figure 9-14 The CosRelationships Module *(Continued)*

```
            readonly attribute unsigned long max_cardinality;
            readonly attribute unsigned long min_cardinality;
            readonly attribute sequence
                    <::CORBA::InterfaceDef> related_object_types;
            Role create_role (in RelatedObject related_object)
                    raises (NilRelatedObject, RelatedObjectTypeError);
    };

    interface RelationshipIterator {
            boolean next_one (out RelationshipHandle rel);
            boolean next_n (in unsigned long how_many,
                            out RelationshipHandles rels);
            void destroy ();
    };

};
```

Figure 9-14 The CosRelationships Module *(Continued)*

Example of Containment Relationships

The example of Figure 9-15 is referred to throughout the following sections to describe roles and relationships. The figure represents two binary, one-to-many containment relationships between a document and a figure and a logo.

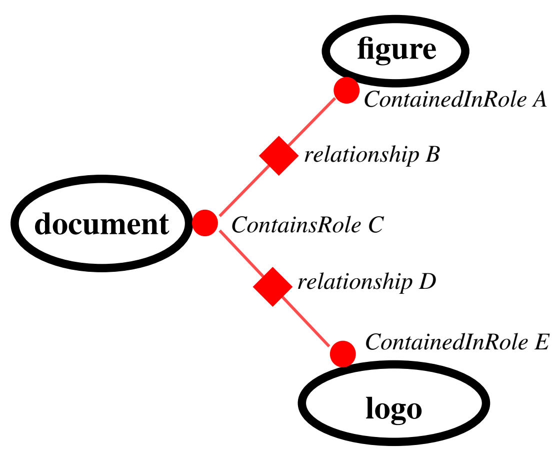


Figure 9-15 Two binary one-to-many containment relationships.

The RelationshipFactory Interface

The *RelationshipFactory* interface defines an operation for creating an instance of a relationship among a set of related objects. The factory also defines two attributes that specify the degree and role types of the relationships it creates.

Creating a Relationship

```
Relationship create (in NamedRoles named_roles)
raises (RoleTypeError,
        MaxCardinalityExceeded,
        DegreeError,
        DuplicateRoleName,
        UnknownRoleName);
```

The create operation creates a new instance of a relationship. The factory is passed a sequence of named roles that represent the related objects in the newly created relationship. The factory, in turn, informs the roles about the new relationship using the link operation described in section .

Roles implement maximum cardinality constraints. A role may refuse to participate in a new relationship because it would violate a cardinality constraint. In such a case, the MaxCardinalityExceeded exception is raised and the offending roles are returned in the exception.

The number of roles passed to the create operation must be the same as the value of the degree attribute. If not, the DegreeError exception is raised.

Role names are used to associate each actual role object with one of the formal roles expected by the relationship to be created.

The set of role names passed to the create operation must be the same as the set of role names in the factory's named_role_types attribute. If not, the UnknowRoleName exception is raised, and the unrecognized names are returned in the exception. The sequence order of the named_roles parameter and the sequence order of the named_role_types need not correspond.

The type of each role passed to the create operation must be of the same type as the type indicated for the corresponding role name in the named_role_types attribute. If not, the RoleTypeError is raised and the offending roles are returned in the exception.

The names of the roles passed to the create operation must be unique within the scope of this relationship type. If not, the DuplicateRoleName exception is raised.

Example of Figure 9-15

The document and the figure were related, that is relationship B was created, by passing roles A and C to the create operation of the relationship factory. Similarly, the document and the logo were related by passing roles C and E to the relationship factory for relationship D.

Determining the Created Relationship's Type

```
readonly attribute ::CORBA::InterfaceDef relationship_type;
```

The relationship created by a factory may be a subtype of the *Relationship* interface. The rrelationship_type attribute indicates the actual types of the relationships created by the factory.

Determining the Degree of a Relationship Type

```
readonly attribute unsigned short degree;
```

The degree attribute indicates the number of roles for the relationships created by the factory.

Example of Figure 9-15

The relationship factory for containment has a degree attribute whose value is 2 because containment is a binary relationship.

Determining Names and Types of the Roles of a Relationship Type

```
readonly attribute NamedRoleTypes named_role_types;
```

The named_role_types attribute indicates the required names and types of roles for the relationships created by the factory. NamedRoleTypes are defined as structures where the role type is given by the CORBA::InterfaceDef for the role objects.

Example of Figure 9-15

The relationship factory for containment has an attribute whose value is a sequence of two CORBA::InterfaceDefs: one for ContainsRole and one for ContainedInRole.

The Relationship Interface

The *Relationship* interface defines an attribute whose value is the named roles of the relationship and an operation to destroy the relationship.

Determining the Roles of a Relationship and Their Names

```
readonly attribute NamedRoles named_roles;
```

The named_roles attribute returns the roles of the relationship. The roles have the names that were indicated in the create operation defined by the *RelationshipFactory* interface.

Example of Figure 9-15

Relationship B has an attribute whose value is a sequence <“A”,InterfaceDef for ContainedInRole; “C”, InterfaceDef for ContainsRole>. Similarly, relationship D has an attribute whose value is a sequence <“E”, InterfaceDef for ContainedInRole; “C”, InterfaceDef for ContainsRole>.

Destroying a Relationship

```
void destroy () raises(CannotUnlink);
```

The destroy operation destroys the relationship between the objects. The roles are unlinked by the relationship implementation before it is destroyed. If roles cannot be unlinked, the CannotUnlink exception is raised and the roles that could not be unlinked are returned in the exception.

Example of Figure 9-15

If destroy is requested of relationship B, the unlink operation is requested of both roles A and C and the relationship B is destroyed.

The Role Interface

The *Role* interface defines operations to:

- navigate the relationship from one role to another,
- enumerate the relationships in which the role participates,
- destroy all relationships in which the role participates,
- link a role to a newly created relationship and
- unlink a role in the destruction process of a relationship and
- destroy the role itself,

Determining the Related Object That a Role Represents

```
readonly attribute RelatedObject related_object;
```

The related_object attribute indicates the related object that the role represents. The related object that the role represents is specified as a parameter to the create operation defined by the *RoleFactory* interface.

Getting Another Related Object

```
RelatedObject get_other_related_object (
        in RelationshipHandle rel,
        in RoleName target_name)
raises (UnknownRoleName,
        UnknownRelationship);
```

The get_other_related_object operation navigates the relationship rel to the related object represented by the role named target_name.

If the role does not know about a role named target_name, the UnknownRoleName exception is raised. If the role does not know about the relationship rel, the UnknownRelationship exception is raised.

Example of Figure 9-15

Assuming role A is named “A”, requesting get_other_related_object(B,”A”) of role C returns the figure. On the other hand, requesting get_other_related_object(D,”E”) of role C returns the logo.

Getting Another Role

```
Role get_other_role (in RelationshipHandle rel,
        in RoleName target_name)
raises (UnknownRoleName, UnknownRelationship);
```

The get_other_role operation navigates the relationship rel to the role named target_name. The role is returned.

If the role does not know about a role named target_name for the relationship rel, the UnknownRoleName exception is raised. If the role does not know about the relationship rel, the UnknownRelationship exception is raised.

Example of Figure 9-15

Assuming role A is named “A”, requesting get_other_role(B,”A”) of role C returns role A. On the other hand, requesting get_other_role(D,”E”) of role C returns role E.

Getting All Relationships in Which a Role Participates

```
void get_relationships (
        in unsigned long how_many,
        out RelationshipHandles rels,
        out RelationshipIterator iterator);
```

The get_relationships operation returns the relationships in which the role participates.

The size of the list is determined by the how_many argument. If there are more relationships than specified by the how_many argument, an iterator is created and returned with the additional relationships. If there are no more relationships, a nil object reference is returned for the iterator. (The *RelationshipIterator* interface is a standard iterator described in the next section.)

Example of Figure 9-15

Requesting get_relationships on role C would return the relationships B and D.

Destroying All Relationships in Which a Role Participates

```
void destroy_relationships()
raises(CannotDestroyRelationship);
```

The destroy_relationships operation destroys all relationships in which the role participates.

The destroy_relationships operation is semantically equivalent to requesting destroy of each relationship in which the role participates. The operation is not required to be implemented in that fashion.

If the destroy_relationships operation cannot destroy one of the relationships, then the CannotDestroyRelationship exception is raised and the relationships that could not be destroyed are returned in the exception.

Example of Figure 9-15

Requesting destroy_relationships of role A causes relationship B to be destroyed. On the other hand, requesting destroy_relationships of role C causes relationships B and D to be destroyed.

Destroying a Role

```
void destroy() raises(ParticipatingInRelationship);
```

The destroy operation destroys the role. The role must not be participating in any relationships. If it is, the ParticipatingInRelationship exception is raised and the relationships in which the role participates are returned in the exception.

Example of Figure 9-15

Requesting destroy_role of role A destroys relationship B and role A.

Checking Minimum Cardinality of a Role

```
boolean check_minimum_cardinality ();
```

The check_minimum_cardinality operation returns *true* if a role satisfies its minimum cardinality constraints. Otherwise, the operation returns *false*.

Example of Figure 9-15

Requesting check_minimum_cardinality of role A would return true since it is participating in relationship B.

Linking a Role in a Newly Created Relationship

```
void link (in RelationshipHandle rel,
        in NamedRoles named_roles)
raises(RelationshipFactory::MaxCardinalityExceeded,
        RelationshipTypeError);
```

Note – The link operation is not intended for general purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship factory create operation.

The link operation informs the role that a new relationship is being created. The role is passed a relationship and a set of named roles that represent related objects in the relationship.

A role can have a maximum cardinality, that is it may limit the number of relationships in which it participates. If the link request would cause the maximum to be exceeded, the MaxCardinalityExceeded exception is raised. If the type of the relationship does not agree with the relationship type that the role expects, the RelationshipTypeError exception is raised.

Example of Figure 9-15

When creating relationship B, the factory for B requested the link (B, A,C) operation on roles A and C. This allows roles A and C to support the navigation and administration operations for relationship B.

Removing a Role from a Relationship

```
void unlink (in RelationshipHandle rel)
raises (UnknownRelationship);
```

Note – The unlink operation is not intended for general purpose clients that create, navigate and destroy relationships. Instead, it is an operation intended for implementations of the relationship destroy operation.

The unlink operation causes the role to delete its record of the relationship.

If the relationship passed as an argument is unknown to the role, the UnknownRelationship exception is raised.

Example of Figure 9-15

The implementation of the destroy operation on relationship B requests unlink(B) of roles A and C. This causes roles A and C to forget their participation in relationship B.

The RoleFactory Interface

The *RoleFactory* interface defines attributes describing the roles that it creates and a single operation to create a role.

Creating a Role

```
Role create_role (in RelatedObject related_object)
        raises (NilRelatedObject, RelatedObjectTypeError);
```

The create_role operation creates a role for the related object passed as a parameter.

A role must represent a related object. If a nil object reference is passed to the factory for the related object, the NilRelatedObject exception is raised.

Role factories can restrict the type of objects the roles they create will represent. If the interface of the related object does not conform, the RelatedObjectTypeError exception is raised.

Example of Figure 9-15

Clients that created roles A, C and E used the create operation of factories that support the *RoleFactory* interface.

Determining the Created Role's Type

```
readonly attribute ::CORBA::InterfaceDef role_type;
```

The role created by a factory may be a subtype of the *Role* interface. The role_type attribute indicates the actual types of the roles created by the factory.

Determining the Maximum Cardinality of a Role

```
readonly attribute unsigned long max_cardinality;
```

The max_cardinality attribute indicates the maximum number of relationships in which a role (created by the factory) participates.

Example of Figure 9-15

The factory for role A returns 1, since a *ContainedIn* role can be in no more than one relationship. Attempts to add role A to more than one relationship result in MaxCardinalityExceeded exceptions. (See the create operation of the *RelationshipFactory* interface and the link operation of the *Role* interface.)

Determining the Minimum Cardinality of a Role

```
readonly attribute unsigned long min_cardinality;
```

The min_cardinality attribute indicates the minimum number of relationships in which a role (created by the factory) participates.

Note, that unlike maximum cardinality, minimum cardinality cannot be enforced since roles will be below their minimum during relationship construction. Roles do support the check_minimum_cardinality operation to report if they are below their minimum.

Example of Figure 9-15

The factory for role A returns 1, since a *ContainedIn* role should be in one relationship.

Determining the Related Object Types for a Role

```
readonly attribute sequence
<::CORBA::InterfaceDef> related_object_types;
```

The factory creates roles that represent related objects in relationships. The related objects must support at least one of the interfaces indicated by the related_object_type attribute.

Example of Figure 9-15

The factory for role C returns the CORBA::InterfaceDef for a document.

The RelationshipIterator Interface

The *RelationshipIterator* interface is returned by the get_relationships operation defined by the *Role* interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one

```
boolean next_one (out RelationshipHandle rel);
```

The next_one operation returns the next relationship; if no more relationships exist, it returns *false*.

next_n

```
boolean next_n (in unsigned long how_many,
        out RelationshipHandles rels);
```

The next_n operation returns at most the requested number of relationships; if no more relationships exist, it returns *false*.

destroy

```
void destroy ();
```

The destroy operation destroys the iterator.

9.4 Graphs of Related Objects

When objects are related using the Relationship Service, *graphs of related objects* are formed. This section focuses on how the Relationship Service supports graphs of related objects. We first describe the graph architecture supported by the service, describe support for traversing the graph and implementing compound operations and then specify the *CosGraphs* module in detail.

Graphs are important for distributed, object-oriented applications. A few examples of graphs are:

Distributed Desktops

Folders and objects are connected together. Folders contain some objects and reference others. Folders may contain or reference other folders. The objects are distributed; they span multiple machines. The distributed desktop is a distributed graph.

Composed Applications

Applications are built out of existing objects that are connected together. An example of such a composed application is a shared white board. The composed application is a graph.

User Interface Hierarchies

Presentation objects visualize semantic objects for users. Presentations contain other presentation objects. For example, a window might contain a button. The user interface hierarchy is a graph.

Compound Documents

A compound document architecture allows graphics, animation, sound, video, etc. to be connected together to give the user the impression of a single document. The compound document is a graph.

9.4.1 Graph Architecture

A graph is a set of nodes and a set of edges, involving those nodes. Nodes are related objects that support the *Node* interface and edges are represented by the relationships that relate nodes.

Figure 9-3 on page 9-9 illustrates an example of a graph.

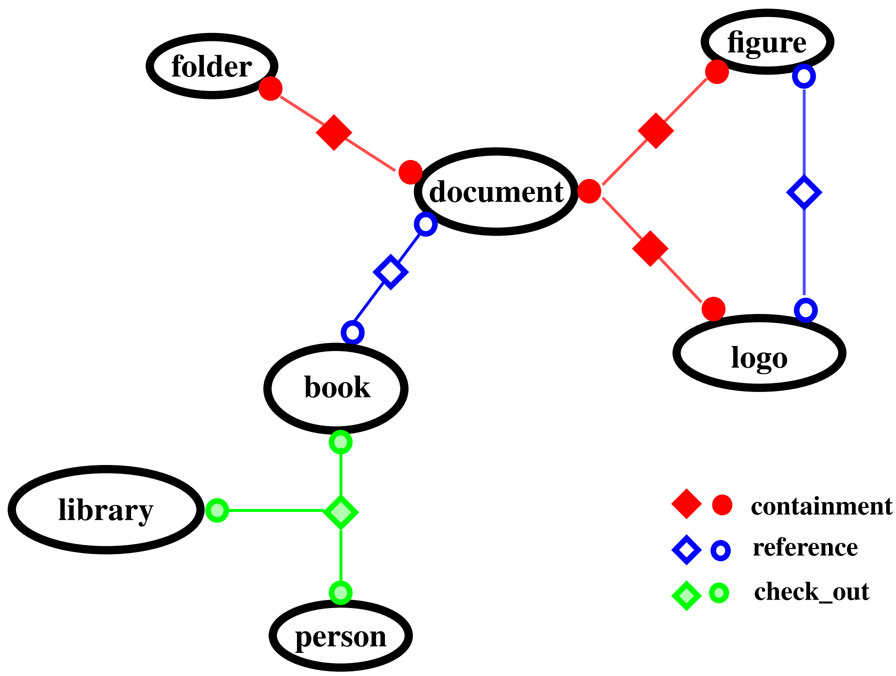


Figure 9-16 An example graph of related objects.

The folder, book, document, figure, library, person and logo are nodes in the graph. The edges of the graph are represented by the relationships:

- containment: the folder and document,
- containment: the document and the figure
- containment: the document and the logo
- reference: the figure and the logo
- reference: the document and the book,
- check_out: the book, the library and the person

The graph architecture supports multiple kinds of relationships. For example, in Figure 9-16, there are containment, reference and check_out relationships. The small circles depict roles for a reference relationship, the solid circles depict roles for a containment relationship and the shaded circles represent the roles of the check_out relationship.

A node can participate in more than one kind of relationship and thus have more than one role. In the example the document has three kinds of roles:

- The *ContainsRole*
- The *ContainedInRole*
- The *ReferencesRole*

Nodes

Nodes are identifiable objects that support the *Node* interface. Nodes collect roles of a related object and the related object itself. A node enables standard traversals of graphs of related objects because it supports the following:

- A readonly attribute defining all of its roles
- An operation allowing roles of a particular type to be returned
- Operations to add and remove roles

The *Node* interface can be inherited by related objects or an object implementing the *Node* interface can be instantiated and interposed in front of related objects. Interposition is particularly useful in these cases:

- When connecting immutable objects, which are objects that are not aware of the Relationship Service
- In order to traverse graphs of related objects without activating the related objects

As such, the *Node* interface defines an attribute whose value is the related object it represents.

9.4.2 Traversing Graphs of Related Objects

The Relationship Service defines a traversal object that, given a starting node, produces a sequence of directed edges of the graph. A directed edge corresponds to a relationship. In particular, it consists of:

- An instance of a relationship,
- A starting node and a starting named role of the edge to indicate direction and
- A sequence containing the remaining nodes and named roles. For binary relationships, there is a single remaining node and role. For n-ary relationships, there are n-1 remaining nodes and roles.

The traversal object works like an iterator, where directed edges are the items being returned.

The traversal object, the nodes and the roles cooperate in traversing the graph. Through the operations of the *Node* interface, the node reveals its roles to the traversal object. Through the operations of the *CosGraphs::Role* interface, a role reveals its directed edges to other nodes. (The *CosGraphs::Role* interface defines an operation allowing a role to reveal directed edges.)

In traversing a graph, the traversal object must detect and represent cycles, and determine the relevant nodes and edges.

Detecting and Representing Cycles

In order to terminate, a traversal must be able to detect a cycle in the graph. In the example of Figure 9-3, the document, the figure, and the logo form a cycle.

To detect cycles in the graph, the traversal object depends on the fact that nodes are identifiable objects, that is they support the *IdentifiableObject* interface defined in section 9.3.6.

To represent cycles in the graph, the traversal object defines a scope of identifiers for the nodes and relationships in the graph. That is, a given traversal *assigns* identifiers to the nodes and relationships that are guaranteed to be unique within the scope of the traversal.

Determining the Relevant Nodes and Edges

A traversal begins at the starting node, emits directed edges and *may* continue to other related nodes. The traversal object is programmable in the criteria it uses for determining the edges to emit and the nodes to visit. The traversal object depends on a "call-back" object supporting the *TraversalCriteria* interface.

Given a node, the traversal criteria computes a sequence of directed edges to include in the traversal. For each edge, the traversal criteria can indicate whether the traversal should continue to an adjacent node. Based on the results of the traversal criteria, the traversal object emits edges and visits other nodes. The process continues until there are no more edges to emit and no more nodes to visit.

Three standard traversal modes are defined to allow clients flexibility in controlling the search order: *depth first, breadth first,* and *best first*. In order to understand the differences between the modes, consider that the traversal maintains an ordered list of the edges which have been produced by visiting nodes. This list initially contains the edges which result from visiting the root node. In each iteration the first edge is removed from the list to be returned and its destination nodes are visited. Depending upon the traversal mode, these edges are: inserted in the beginning of the list (depth first), appended to the end of the list (breadth first), or inserted into the list which is sorted by the edge's weight (best first).

9.4.3 Compound Operations

Traversal objects are especially important in implementing compound operations on graphs of related objects. By compound operations, we mean operations that apply to some subset of the nodes and edges in the graph. Examples of compound operations include operations, such as copy, move, remove, externalize, print, and so forth.

Note – The Relationship Service defines a framework for compound operations but does not define specific compound operations. The Life Cycle and the Externalization Service specifications define compound operations that depend on the Relationship Service.

A compound operation may be implemented either in one or two passes. A compound operation implemented in one pass traverses the graph itself and applies the operation as it proceeds.

A compound operation implemented in two passes uses the traversal object defined by the Relationship Service to determine the relevant nodes and detect and represent cycles. The second pass simply applies the operation to the results of the first pass.

A compound operation implemented in two passes provides a *TraversalCriteria* object for the traversal service.

9.4.4 An Example Traversal Criteria

Consider a traversal of a graph with a traversal criteria object that uses propagation values defined by the relationships to determine whether to emit an edge and whether to proceed to another node. The traversal criteria is given a node by the traversal. The traversal criteria then requests propagation values from each of the node's roles.

Figure 9-17 illustrates a traversal of a graph using a traversal criteria for a compound copy operation. Using the propagation_for operation defined by *CompoundLifeCycle::Role* interface, the traversal criteria obtains the propagation value for the copy operation from each of the node's roles.

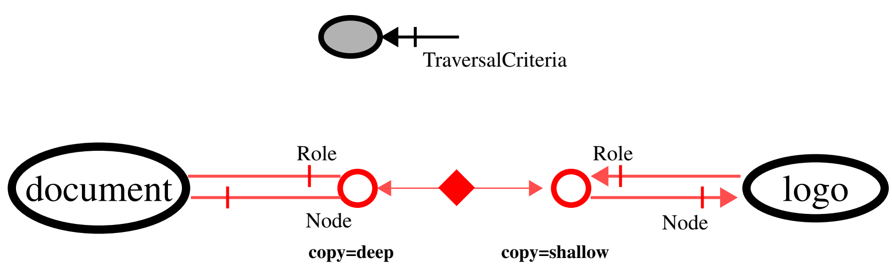


Figure 9-17 A traversal of a graph for compound copy operation.

Propagation

Compound operations may propagate from one node to another depending on the semantics of the relationship between the nodes. The propagation semantics of a relationship depend on the direction the relationship is being traversed. A propagation value is either *deep*, *shallow*, *inhibit* or *none*.

Deep means that the operation is applied to the node, to the relationship and to the related objects. In the example of Figure 9-17, the propagation value for the copy operation is deep from the document to the logo; the copy propagates from the document to the logo across the containment relationship. The traversal criteria for copy that encounters a deep propagation value would instruct the traversal object to emit the edge and visit the logo.

Shallow means that the operation is applied to the relationship but not to the related objects. In the example of Figure 9-17, the propagation value for the copy operation from the logo to the document is shallow. The traversal criteria for copy that encounters a shallow propagation value would instruct the traversal object to emit the edge but the document is not visited.

None means that the operation has no effect on the relationship and no effect on the related objects. A traversal criteria that encounters a none propagation value would not return any edges and related nodes are not visited.

Figure 9-18 summarizes how deep, shallow and node propagation values affect nodes, roles and relationships.

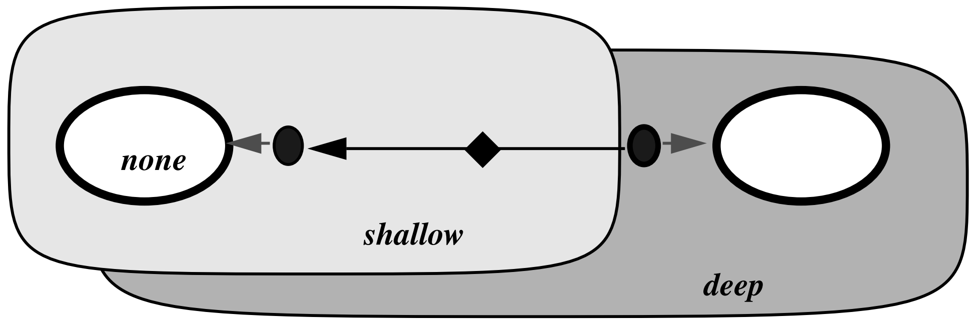


Figure 9-18 How deep, shallow and none propagation values affect nodes, roles and relationships.

Inhibit means that the operation should not propagate to the node via any of the node's roles. Inhibit is particularly meaningful for the remove operation to provide so-called "existence-ensuring relationships".

For more discussion of propagation values, see [1.].

9.4.5 The CosGraphs Module

The CosGraphs module defines the support for graphs of related objects. It defines the following interfaces:

- *TraversalFactory* interface for creating traversal objects
- *Traversal* interface for enumerating directed edges of a graph,
- *TraversalCriteria* "call-back" interface to allow programmability of the traversal object
- *Node* interface for collecting the roles of a related object
- *NodeFactory* interface for creating nodes
- *Role* interface to support traversals

The CosGraphs module is shown in Figure 9-14.

```
#include <Relationships.idl>
#include <ObjectIdentity.idl>

module CosGraphs {

	interface TraversalFactory;
	interface Traversal;
	interface TraversalCriteria;
	interface Node;
	interface NodeFactory;
	interface Role;
	interface EdgeIterator;

	struct NodeHandle {
		Node the_node;
		::CosObjectIdentity::ObjectIdentifier constant_random_id;
	};
	typedef sequence<NodeHandle> NodeHandles;

	struct NamedRole {
		Role the_role;
		::CosRelationships::RoleName the_name;
	};
	typedef sequence<NamedRole> NamedRoles;

		struct EndPoint {
			NodeHandle the_node;
			NamedRole the_role;
		};
		typedef sequence<EndPoint> EndPoints;

		struct Edge {
			EndPoint from;
			::CosRelationships::RelationshipHandle the_relationship;
			EndPoints relatives;
		};
		typedef sequence<Edge> Edges;

	enum PropagationValue {deep, shallow, none, inhibit};
	enum Mode {depthFirst, breadthFirst, bestFirst};

	interface TraversalFactory {
			Traversal create_traversal_on (
					in NodeHandle root_node,
					in TraversalCriteria the_criteria,
					in Mode how);
	};
```

Figure 9-19 The CosGraphs Module

```
interface Traversal {
        typedef unsigned long TraversalScopedId;
        struct ScopedEndPoint {
            EndPoint point;
            TraversalScopedId id;
        };
        typedef sequence<ScopedEndPoint> ScopedEndPoints;
        struct ScopedRelationship {
            ::CosRelationships::RelationshipHandle
                scoped_relationship;
            TraversalScopedId id;
        };
        struct ScopedEdge {
            ScopedEndPoint from;
            ScopedRelationship the_relationship;
            ScopedEndPoints relatives;
        };
        typedef sequence<ScopedEdge> ScopedEdges;
        boolean next_one (out ScopedEdge the_edge);
        boolean next_n (in short how_many,
                out ScopedEdges the_edges);
        void destroy ();
};

interface TraversalCriteria {
        struct WeightedEdge {
            Edge the_edge;
            unsigned long weight;
            sequence<NodeHandle> next_nodes;
        };
        typedef sequence<WeightedEdge> WeightedEdges;
        void visit_node(in NodeHandle a_node,
                in Mode search_mode);
        boolean next_one (out WeightedEdge the_edge);
        boolean next_n (in short how_many,
                out WeightedEdges the_edges);
        void destroy();
};
```

Figure 9-19 The CosGraphs Module *(Continued)*

```
interface Node: ::CosObjectIdentity::IdentifiableObject {
    typedef sequence<Role> Roles;
    exception NoSuchRole {};
    exception DuplicateRoleType {};

        readonly attribute ::CosRelationships::RelatedObject
            related_object;
        readonly attribute Roles roles_of_node;
        Roles roles_of_type (
                in ::CORBA::InterfaceDef role_type);
        void add_role (in Role a_role)
                raises (DuplicateRoleType);
        void remove_role (in ::CORBA::InterfaceDef of_type)
            raises (NoSuchRole);
};

interface NodeFactory {
        Node create_node (in Object related_object);
};

interface Role : ::CosRelationships::Role {
        void get_edges ( in long how_many,
                out Edges the_edges,
                out EdgeIterator the_rest);
};

interface EdgeIterator {
        boolean next_one (out Edge the_edge);
        boolean next_n ( in unsigned long how_many,
                out Edges the_edges);
        void destroy ();
};

};
```

Figure 9-19 The CosGraphs Module *(Continued)*

The TraversalFactory Interface

The *TraversalFactory* interface creates traversal objects. The *Traversal* interface is used by clients that want to traverse graphs of related objects according to some traversal criteria.

create_traversal_on

```
Traversal create_traversal_on (
    in NodeHandle root_node,
    in TraversalCriteria the_criteria,
    in Mode how);
```

The create_traversal_on operation creates a traversal object starting at the root_node. The created traversal object uses the *TraversalCriteria* object to determine which directed edges to emit and which nodes to visit. The mode parameter indicates whether the traversal will proceed in a depth first, breadth first or best first fashion.

The Traversal Interface

Traversal objects iterate through ScopedEdges of the graph according to the traversal criteria and the mode established when the traversal was created. The traversal also defines a scope for the nodes and edges it returns; that is, it assigns identifiers to the nodes and edges it returns. The identifiers are unique within the scope of a given traversal. ScopedEdges are given by the following structure:

```
struct ScopedEdge {
ScopedEndPoint from;
ScopedRelationship the_relationship;
ScopedEndPoints relatives;
};
typedef sequence<ScopedEdge> ScopedEdges;
```

A ScopedEdge consists of a distinguished scoped end point, a scoped relationship and a sequence of scoped end points. The distinguished scoped end point indicates the direction of the edge. The scoped end point consists of a node, a role, and an identifier for the node that is unique within the scope of the traversal.

next_one

```
boolean next_one (out ScopedEdge the_edge);
```

The next_one operation returns the next scoped edge; if no more scoped edges exist, it returns *false*.

next_n

```
boolean next_n (in short how_many,
        out ScopedEdges the_edges);
```

The next_n operation returns at most the requested number of scoped edges.

destroy

```
void destroy ();
```

The destroy operation destroys the traversal.

The TraversalCriteria Interface

The *TraversalCriteria* interface is used by the traversal object to determine which edges to emit and which nodes to visit from a given node. The traversal criteria behaves like an iterator of weighted edges. Weighted edges are given by the following structure:

```
struct WeightedEdge {
Edge the_edge;
unsigned long weight;
sequence<NodeHandle> next_nodes;
};
typedef sequence<WeightedEdge> WeightedEdges;
```

A WeightedEdge consists of an edge, a weight and a sequence of nodes indicating if the traversal should continue to the nodes. The weight is only meaningful for the best first traversal.

next_one

```
boolean next_one (out WeightedEdge the_edge);
```

The next_one operation returns the next weighted edge; if no more weighted edges exist, it returns *false*.

next_n

```
boolean next_n (in short how_many,
        out WeightedEdges the_edges);
```

The next_n operation returns at most the requested number of weighted directed edges.

destroy

```
void destroy();
```

The destroy operation destroys the traversal criteria.

visit_node

```
void visit_node(in NodeHandle a_node,
        in Mode search_mode);
```

The visit_node operation establishes the node for which the traversal criteria will iterate and indicates the current search mode. As the traversal object traverses the graph, it visits nodes by requesting the visit_node operation of the traversal criteria, followed by next_one/next_n requests to obtain the outgoing edges from the node.

For depthFirst and breadthFirst modes, the weight field in the weighted edges is ignored. In the bestFirst mode, the weight value is utilized to order the traversal's edges list which is sorted by this value in ascending order.

If weighted edges from a previous node remain when visit_node is requested, the traversal criteria discards the previous edges.

The Node Interface

The *Node* interface defines operations that are useful in navigating graphs of related objects. In particular, it defines:

- Areadonly attribute giving all of the node's roles
- An operation allowing roles conforming to a particular type to be returned
- Operations to add and remove roles

Roles are distinguished in nodes in the OMG IDL of their interfaces.

A node cannot posses two roles where one role is a subtype of the other. This is precluded by the add_role operation.

A node can posses two or more roles that have a common supertype. The set of roles can be obtained by passing the common supertype to the roles_of_type operation.

related_object

```
readonly attribute ::CosRelationships::RelatedObject
 related_object;
```

The related_object attribute gives the related object that the node represents. This is useful when relating immutable objects.

roles_of_node

```
readonly attribute Roles roles_of_node;
```

The roles_of_node attribute gives all of the node's roles.

roles_of_type

```
Roles roles_of_type (
        in ::CORBA::InterfaceDef role_type);
```

The roles_of_type operation returns the node's roles that conform to the role_type parameter. A role conforms to role_type if it's interface is the same or is a subtype of role_type.

add_role

```
void add_role (in Role a_role)
raises (DuplicateRoleType);
```

The add_role operation adds a role to the node. If the node posses a role of the same type, a supertype or a subtype of a_role, the DuplicateRoleType exception is raised.

remove_role

```
void remove_role (in ::CORBA::InterfaceDef of_type)
raises (NoSuchRole);
```

The remove_role operation removes all the roles that conform to the of_type parameter. If no roles conform to the of_type parameter, the NoSuchRole exception is raised.

The NodeFactory Interface

The *NodeFactory* interface defines a single operation for creating nodes.

create_node

```
Node create_node (in Object related_object);
```

The create_node operation creates a node whose related_object attribute is initialized to the related_object parameter.

The Role Interface

The *CosGraphs::Role* interface extends the *CosRelationships::Role* interface with a single operation to return a role's view of it's relationships. The role's view of a relationship is given by the following Edge structure:

```
struct Edge {
EndPoint from;
::CosRelationships::RelationshipHandle the_relationship;
EndPoints relatives;
};
typedef sequence<Edge> Edges;
```

The edge structure is defined by an end point, a relationship and the other end points. The from end point is the role and its related object.

get_edges

```
void get_edges ( in long how_many,
        out Edges the_edges,
        out EdgeIterator the_rest);
```

The get_edges operation returns the edges in which the role participates.

The size of the list is determined by the how_many argument. If there are more edges than specified by the how_many argument, an iterator is created and returned. If there are no more edges, a nil object reference is returned for the iterator.

The EdgeIterator Interface

The *EdgeIterator* interface is returned by the get_edges operation defined by the *CosGraphs::Role* interface. It allows clients to iterate through any additional relationships in which the role participates.

next_one

```
boolean next_one (out Edge the_edge);
```

The next_one operation returns the next edge; if no more edges exist, it returns *false*.

next_n

```
boolean next_n ( in unsigned long how_many,
        out Edges the_edges);
```

The next_n operation returns at most the requested number of edges.

destroy

```
void destroy ();
```

The destroy operation destroys the iterator.

9.5 Specific Relationships

The Relationship Service defines two important relationships, *containment* and *reference* as part of its specification. The example used throughout this specification has been in terms of these two relationships.

9.5.1 Containment and Reference

Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment and reference are examples of relationships. However, since containment and reference are very common relationships, the Relationship Service defines them as standard.

Containment is defined by interfaces for a relationship and two roles: the *CosContainment::Relationship* interface, the *CosContainment::ContainsRole* interface, and the *CosContainment::ContainedInRole* interface. *Relationship* is a subtype of *CosRelationships::Relationship* and *ContainedInRole* and *ContainsRole* are subtypes of *CosGraphs::Role*.

Similarly, reference is defined by interfaces for a relationship and two roles: the *CosReference::Relationship* interface, the *CosReference::ReferencesRole* interface, and the *CosReference::ReferencedByRole* interface. *Relationship* is a subtype of *CosRelationships::Relationship* and *ReferencesRole* and *ReferencedByRole* are subtypes of *CosGraphs::Role*.

9.5.2 The CosContainment Module

The *CosContainment* module is shown in Figure 9-14.

```
#include <Graphs.idl>

module CosContainment {

    interface Relationship :
        ::CosRelationships::Relationship {};

    interface ContainsRole : ::CosGraphs::Role {};

    interface ContainedInRole : ::CosGraphs::Role {};

};
```

Figure 9-20 The CosContainment Module

The *CosContainment* module does not define new operations. It introduces new IDL types to represent containment. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	value
relationship_type	CosContainment::Relationship
degree	2
named_role_types	“ContainsRole”,CosContainment::ContainsRole; “ContainedInRole”,CosContainment::ContainedInRole

The CosRelationships::RelationshipFactory::create operation will raise DegreeError if the number of roles passed as arguments is not 2. It will raise RoleTypeError if the roles are not *CosContainment*::*ContainsRole* and *CosContainment*::*ContainedInRole*. It will raise MaxCardinalityExceeded if the *CosContainment*::*ContainedInRole* is already participating in a relationship.

RoleFactory attribute for ContainsRole	value
role_type	CosContainment::ContainsRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosContainment::Relationship* interface.

RoleFactory attribute for ContainedInRole	value
role_type	CosContainment::ContainedInRole
maximum_cardinality	1
minimum_cardinality	1
related_object_types	CosGraphs::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosContainment::Relationship* interface. The *CosRelationships::RoleFactory::*link operation will raise MaxCardinalityExceeded if it is already participating in a containment relationship.

9.5.3 The CosReference Module

The *CosReference* module is given in Figure 9-21.

```
#include <Graphs.idl>

module CosReference {

    interface Relationship :
        ::CosRelationships::Relationship {};
```

Figure 9-21 The CosReference Module

```
        interface ReferencesRole : CosGraphs::Role {};

        interface ReferencedByRole : ::CosGraphs::Role {};

};
```

Figure 9-21 The CosReference Module *(Continued)*

The *CosReference* module does not define new operations. It introduces new IDL types to represent reference. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
relationship_type	CosReference::Relationship
degree	2
named_role_types	“ReferencesRole”,CosReference::ReferencesRole; “ReferencedByRole”,CoReference::ReferencedByRole

The CosRelationships::RelationshipFactory::create operation will raise DegreeError if the number of roles passed as arguments is not 2. It will raise RoleTypeError if the roles are not *CosReference::ReferencesRole* and *CosReference*::*ReferencedByRole*.

RoleFactory attribute for ReferencesRole	**value**
role_type	CosReference::ReferencesRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosReference::Relationship* interface.

RoleFactory attribute for ReferencedByRole	**value**
role_type	CosReference::ReferencedByRole
maximum_cardinality	unbounded
minimum_cardinality	0
related_object_types	CosGraphs::Node

The *CosRelationships::RoleFactory::*create_role operation will raise the RelatedObjectTypeError if the related object passed as a parameter does not support the *CosGraphs::Node* interface. The *CosRelationships::RoleFactory::*link operation will raise RelationshipTypeError if the rel parameter does not conform to the *CosRelationship::Relationship* interface.

9.6 References

1. James Rumbaugh, “Controlling Propagation of Operations using Attributes on Relations.” *OOPSLA 1988 Proceedings*, pg. 285-296.
2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, “Object-oriented Modeling and Design.” Prentice Hall, 1991.

Transaction Service Specification 10

This chapter provides the following information about the Transaction Service:

- A description of the service, which explains the functional, design, and performance requirements that are satisfied by this specification.
- An overview of the Transaction Service that introduces the concepts used throughout this chapter.
- A description of the Transaction Service's architecture and a detailed definition of the Transaction Service, including definitions of its interfaces and operations.
- A user's view of the Transaction Service as seen by the application programmer, including client and object implementer.
- An implementer's view of the Transaction Service, which will interest Transaction Service and ORB providers.

This chapter also contains an appendix that explains the relationship between the Transaction Service and TP standards, and an appendix that contains transaction terms.

Contents

This chapter contains the following sections.

Section Title	**Page**
"Service Description	10-2
"Service Architecture	10-12
"Transaction Service Interfaces	10-17
"The User's View	10-34
"The Implementers' View	10-48

Section Title	Page
"The CosTransactions Module	10-69
Appendix A "Relationship of Transaction Service to TP Standards	10-74
Appendix B "Transaction Service Glossary	10-85

10.1 Service Description

The concept of transactions is an important programming paradigm for simplifying the construction of reliable and available applications, especially those that require concurrent access to shared data. The transaction concept was first deployed in commercial operational applications where it was used to protect data in centralized databases. More recently, the transaction concept has been extended to the broader context of distributed computation. Today it is widely accepted that transactions are the key to constructing reliable distributed applications.

The Transaction Service described in this specification brings the transaction paradigm, essential to developing reliable distributed applications, and the object paradigm, key to productivity and quality in application development, together to address the business problems of commercial transaction processing.

10.1.1 Overview of Transactions

The Transaction Service supports the concept of a transaction. A transaction is a unit of work that has the following (ACID) characteristics:

- A transaction is **atomic**; if interrupted by failure, all effects are undone (rolled back).
- A transaction produces **consistent** results; the effects of a transaction preserve invariant properties.
- A transaction is **isolated**; its intermediate states are not visible to other transactions. Transactions appear to execute serially, even if they are performed concurrently.
- A transaction is **durable**; the effects of a completed transaction are persistent; they are never lost (except in a catastrophic failure).

A transaction can be terminated in two ways: the transaction is either committed or rolled back. When a transaction is committed, all changes made by the associated requests are made permanent. When a transaction is rolled back, all changes made by the associated requests are undone.

The Transaction Service defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of (noncatastrophic) failure. No requirements are placed on the objects other than those defined by the Transaction Service interfaces.

Transaction semantics can be defined as part of any object that provides ACID properties. Examples are ODBMSs and persistent objects. The value of a separate transaction service is that it allows:

- Transactions to include multiple, separately defined, ACID objects.
- The possibility of transactions which include objects and resources from the non-object world.

10.1.2 Transactional Applications

The Transaction Service provides transaction synchronization across the elements of a distributed client/server application.

A transaction can involve multiple objects performing multiple requests. The scope of a transaction is defined by a transaction context that is shared by the participating objects. The Transaction Service places no constraints on the number of objects involved, the topology of the application or the way in which the application is distributed across a network.

In a typical scenario, a client first begins a transaction (by issuing a request to an object defined by the Transaction Service), which establishes a transaction context associated with the client thread. The client then issues requests. These requests are implicitly associated with the client's transaction; they share the client's transaction context. Eventually, the client decides to end the transaction (by issuing another request). If there were no failures, the changes produced as a consequence of the client's requests would then be committed; otherwise, the changes would be rolled back.

In this scenario, the transaction context is transmitted implicitly to the objects, without direct client intervention—See "Application Programming Models" on page 10-34. The Transaction Service also supports scenarios where the client directly controls the propagation of the transaction context. For example, a client can pass the transaction context to an object as an explicit parameter in a request. An implementation of the Transaction Service might limit the client's ability to explicitly propagate the transaction context, in order to guarantee transaction integrity (See "Application Programming Models" on page 10-34, Subsection "Direct Context Management: Explicit Propagation").

The Transaction Service does not require that all requests be performed within the scope of a transaction. A request issued outside the scope of a transaction has no associated transaction context. It is up to each object to determine its behavior when invoked outside the scope of a transaction; an object that requires a transaction context can raise a standard exception.

10.1.3 Definitions

Applications supported by the Transaction Service consist of the following entities:

- Transactional Client (TC)
- Transactional Objects (TO)

- Recoverable Objects
- Transactional Servers
- Recoverable Servers

The following figure shows a simple application which includes these basic elements.

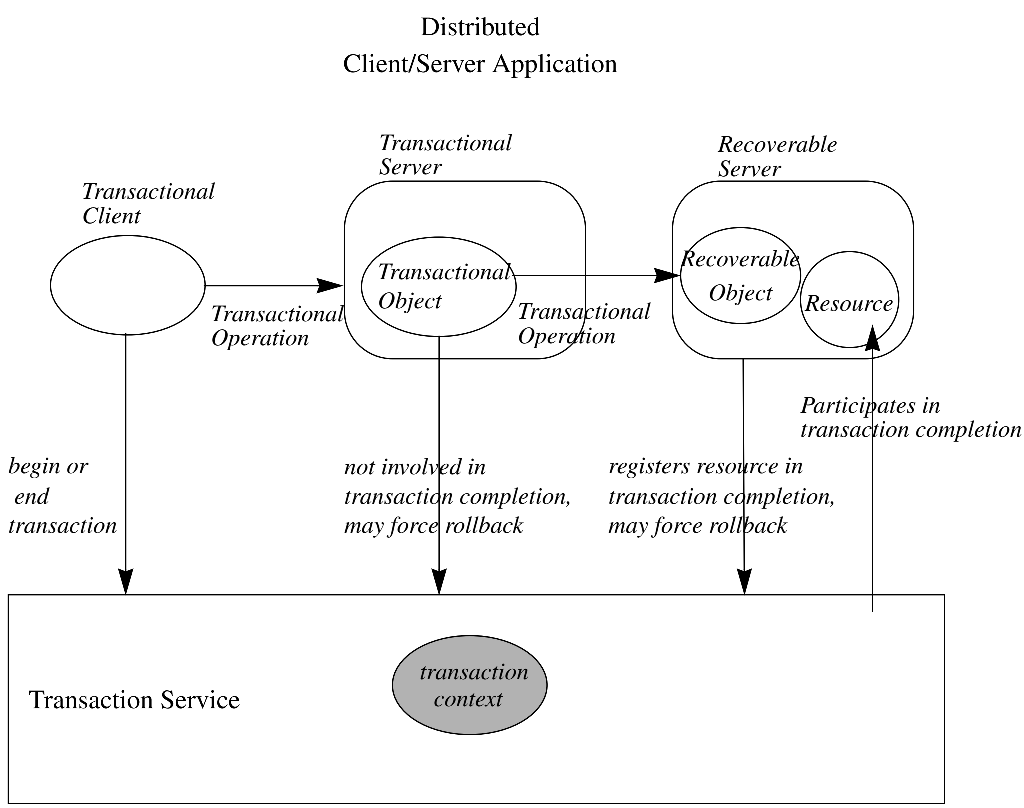


Figure 10-1 Application Including Basic Elements

Transactional Client

A transactional client is an arbitrary program that can invoke operations of many transactional objects in a single transaction.

The program that begins a transaction is called the transaction originator.

Transactional Object

We use the term *transactional object* to refer to an object whose behavior is affected by being invoked within the scope of a transaction. A transactional object typically contains or indirectly refers to persistent data that can be modified by requests.

The Transaction Service does not require that all requests have transactional behavior, even when issued within the scope of a transaction. An object can choose to not support transactional behavior, or to support transactional behavior for some requests but not others.

We use the term *nontransactional object* to refer to an object none of whose operations are affected by being invoked within the scope of a transaction.

If an object does not support transactional behavior for a request, then the changes produced by the request might not survive a failure and the changes will not be undone if the transaction associated with the request is rolled back.

An object can also choose to support transactional behavior for some requests but not others. This choice can be exercised by both the client and the server of the request.

The Transaction Service permits an interface to have both transactional and nontransactional implementations. No IDL extensions are introduced to specify whether or not an operation has transactional behavior. Transactional behavior can be a quality of service that differs in different implementations.

Transactional objects are used to implement two types of application servers:

- Transactional Server
- Recoverable Server

Recoverable Objects and Resource Objects

To implement transactional behavior, an object must participate in certain protocols defined by the Transaction Service. These protocols are used to ensure that all participants in the transaction agree on the outcome (commit or rollback) and to recover from failures.

To be more precise, an object is required to participate in these protocols only if it directly manages data whose state is subject to change within a transaction. An object whose data is affected by committing or rolling back a transaction is called a recoverable object.

A recoverable object is by definition a transactional object. However, an object can be transactional but not recoverable by implementing its state using some other (recoverable) object. A client is concerned only that an object is transactional; a client cannot tell whether a transactional object is or is not a recoverable object.

A recoverable object must participate in the Transaction Service protocols. It does so by registering an object called a *Resource* with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

A recoverable object typically involves itself in a transaction because it is required to retain in stable storage certain information at critical times in its processing. When a recoverable object restarts after a failure, it participates in a recovery protocol based on the contents (or lack of contents) of its stable storage.

A transaction can be used to coordinate non-durable activities which do not require permanent changes to storage.

Transactional Server

A transactional server is a collection of one or more objects whose behavior is affected by the transaction, but which have no recoverable states of their own. Instead, it implements transactional changes using other recoverable objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to be rolled back.

Recoverable Server

A recoverable server is a collection of objects, at least one of which is recoverable.

A recoverable server participates in the protocols by registering one or more *Resource* objects with the Transaction Service. The Transaction Service drives the commit protocol by issuing requests to the resources registered for a transaction.

10.1.4 Transaction Service Functionality

The Transaction Service provides operations to:

- Control the scope and duration of a transaction
- Allow multiple objects to be involved in a single, atomic transaction
- Allow objects to associate changes in their internal state with a transaction
- Coordinate the completion of transactions

Transaction Models

The Transaction Service supports two distributed transaction models: flat transactions and nested transactions. An implementation of the Transaction Service is not required to support nested transactions.

Flat Transactions

The Transaction Service defines support for a flat transaction model. The definition of the function provided, and the commitment protocols used, is modelled on the X/Open DTP transaction model definition.[1]

A flat transaction is considered to be a top-level transaction—see the next section—that cannot have a child transaction.

---

1. See *Distributed Transaction Processing: The XA Specification,* X/Open Document C193. X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.

Nested Transactions

The Transaction Service also defines a nested transaction model. Nested transactions provide for a finer granularity of recovery than flat transactions. The effect of failures that require rollback can be limited so that unaffected parts of the transaction need not rollback.

Nested transactions allow an application to create a transaction that is embedded in an existing transaction. The existing transaction is called the ***parent*** of the subtransaction; the subtransaction is called a ***child*** of the parent transaction.

Multiple subtransactions can be embedded in the same parent transaction. The children of one parent are called ***siblings***.

Subtransactions can be embedded in other subtransactions to any level of nesting. The ***ancestors*** of a transaction are the parent of the subtransaction and (recursively) the parents of its ancestors. The ***descendants*** of a transaction are the children of the transaction and (recursively) the children of its descendants.

A top-level transaction is one with no parent. A top-level transaction and all of its descendants are called a ***transaction family***.

A subtransaction is similar to a top-level transaction in that the changes made on behalf of a subtransaction are either committed in their entirety or rolled back. However, when a subtransaction is committed, the changes remain contingent upon commitment of all of the transaction’s ancestors.

Subtransactions are strictly nested. A transaction cannot commit unless all of its children have completed. When a transaction is rolled back, all of its children are rolled back.

Objects that participate in transactions must support isolation of transactions. The concept of isolation applies to subtransactions as well as to top level transactions. When a transaction has multiple children, the children appear to other transactions to execute serially, even if they are performed concurrently.

Subtransactions can be used to isolate failures. If an operation performed within a subtransaction fails, only the subtransaction is rolled back. The parent transaction has the opportunity to correct or compensate for the problem and complete its operation. Subtransactions can also be used to perform suboperations of a transaction in parallel, without the risk of inconsistent results.

Transaction Termination

A transaction is terminated by issuing a request to commit or rollback the transaction. Typically, a transaction is terminated by the client that originated the transaction—the transaction originator. Some implementations of the Transaction Service may allow transactions to be terminated by Transaction Service clients other than the one which created the transaction.

Any participant in a transaction can force the transaction to be rolled back (eventually). If a transaction is rolled back, all participants rollback their changes. Typically, a participant may request the rollback of the current transaction after encountering a failure. It is implementation-specific whether the Transaction Service itself monitors the participants in a transaction for failures or inactivity.

Transaction Integrity

Some implementations of the Transaction Service impose constraints on the use of the Transaction Service interfaces in order to guarantee integrity equivalent to that provided by the interfaces which support the X/Open DTP transaction model. This is called *checked* transaction behavior.

For example, allowing a transaction to commit before all computations acting on behalf of the transaction have completed can lead to a loss of data integrity. Checked implementations of the Transaction Service will prevent premature commitment of a transaction.

Other implementations of the Transaction Service may rely completely on the application to provide transaction integrity. This is called *unchecked* transaction behavior.

Transaction Context

As part of the environment of each ORB-aware thread, the ORB maintains a transaction context. The transaction context associated with a thread is either null (indicating that the thread has no associated transaction) or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time, in the same execution environment or in multiple execution environments.

The transaction context can be implicitly transmitted to transactional objects as part of a transactional operation invocation. The Transaction Service also allows programmers to pass a transaction context as an explicit parameter of a request.

Synchronization

The Transaction Service defines support for a synchronization interface. This provides a protocol by which an object may be notified prior to the start of the two-phase commit protocol within the coordinator with which it is registered. An implementation of the Transaction Service is not required to support synchronization.

10.1.5 Principles of Function, Design, and Performance

The Transaction Service defined in this specification fulfills a number of functional, design, and performance requirements.

Functional Requirements

The Transaction Service defined in this specification addresses the following functional requirements:

Support for multiple transaction models. The flat transaction model, which is widely supported in the industry today, is a mandatory component of this specification. The nested transaction model, which provides finer granularity isolation and facilitates object reuse in a transactional environment, is an optional component of this specification.

Evolutionary Deployment. An important property of object technology is the ability to “wrapper” existing programs (coarse grain objects) to allow these functions to serve as building blocks for new business applications. This technique has been successfully used to marry object-oriented end-user interfaces with commercial business logic implemented using classical procedural techniques.

It can similarly be used to encapsulate the large body of existing business software on legacy environments and leverage that in building new business applications. This will allow customers to gradually deploy object technology into their existing environments, without having to reimplement all existing business functions.

Model Interoperability. Customers desire the capability to add object implementations to existing procedural applications and to augment object implementations with code that uses the procedural paradigm. To do so in a transaction environment requires that a single transaction be shared by both the object and procedural code. This includes the following:

- A single transaction which includes ORB and non-ORB applications and resources.
- Interoperability between the object transaction service model and the X/Open Distributed Transaction Processing (DTP) model.
- Access to existing (non-object) programs and resource managers by objects.
- Access to objects by existing programs and resource managers.
- Coordination by a single transaction service of the activities of both object and non-object resource managers.
- The network case: A single transaction, distributed between an object and non-object system, each of which has its own Transaction Service.

The Transaction Service accommodates this requirement for implementations where interoperability with X/Open DTP-compliant transactional applications is necessary.

Network Interoperability. Customers require the ability to interoperate between systems offered by multiple vendors:

- Single transaction service, single ORB - It must be possible for a single transaction service to interoperate with itself using a single ORB.
- Multiple transaction services, single ORB - It must be possible for one transaction service to interoperate with a cooperating transaction service using a single ORB.
- Single transaction service, multiple ORBs - It must be possible for a single transaction service to interoperate with itself using different ORBs.

- Multiple transaction services, multiple ORBs - It must be possible for one transaction service to interoperate with a cooperating transaction service using different ORBs.

The Transaction Service specifies all required interactions between cooperating Transaction Service implementations necessary to support a single ORB. The Transaction Service depends on ORB interoperability (as defined by the CORBA specification) to provide cooperating Transaction Services across different ORBs.

Flexible transaction propagation control. Both client and object implementations can control transaction propagation:

- A client controls whether or not its transaction is propagated with an operation.
- A client can invoke operations on objects with transactional behavior and objects without transactional behavior within the scope of a single transaction.
- An object can specify transactional behavior for its interfaces.

The Transaction Service supports both implicit (system-managed) propagation and explicit (application-managed) propagation. With implicit propagation, transactional behavior is not specified in the operation's signature. With explicit propagation, applications define their own mechanisms for sharing a common transaction.

Support for TP Monitors. Customers need object technology to build mission-critical applications. These applications are deployed on commercial transaction processing systems where a TP Monitor provides both efficient scheduling and the sharing of resources by a large number of users. It must be possible to implement the Transaction Service in a TP monitor environment. This includes:

- The ability to execute multiple transactions concurrently.
- The ability to execute clients, servers, and transaction services in separate processes.

The Transaction Service is usable in a TP Monitor environment.

Design Requirements

The Transaction Service supports the following design requirements:

Exploitation of OO Technology. This specification permits a wide variety of ORB and Transaction Service implementations and uses objects to enable ORB-based, secure implementations. The Transaction Service provides the programmer with easy to use interfaces that hide some of the complexity inherent in general-use specifications. Meaningful user applications can be constructed using interfaces that are as simple or simpler than their procedural equivalents.

Low Implementation Cost. The Transaction Service specification considers cost from the perspective of three users of the service - clients, ORB implementers, and Transaction Service providers.

- For clients, it allows a range of implementations which are compliant with the proposed architecture. Many ORB implementations will exist in client workstations which have no requirement to understand transactions within themselves, but will find it highly desirable to interoperate with server platforms that implement transactions.
- The specification provides for minimal impact to the ORB. Where feasible, function is assigned to an object service implementation to permit the ORB to continue to provide high performance object access when transactions are not used.
- Since this Transaction Service will be supported by existing (procedural) transaction managers, the specification allows implementations that reuse existing procedural Transaction Managers.

Portability. The Transaction Service specification provides for portability of applications. It also defines an interface between the ORB and the Transaction Service that enables individual Transaction Service implementations to be ported between different ORB implementations.

Avoidance of OMG IDL interface variants. The Transaction Service allows a single interface to be supported by both transactional and non-transactional implementations. This approach avoids a potential "combinatorial explosion" of interface variants that differ only in their transactional characteristics. For example, the existing Object Service interfaces can support transactional behavior without change.

Support for both single-threaded and multi-threaded implementations. The Transaction Service defines a flexible model that supports a variety of programming styles. For example, a client with an active transaction can make requests for the same transaction on multiple threads. Similarly, an object can support multiple transactions in parallel by using multiple threads.

A wide spectrum of implementation choices. The Transaction Service allows implementations to choose the degree of checking provided to guarantee legal behavior of its users. This permits both robust implementations which provide strong assurances for transaction integrity and lightweight implementations where such checks are not warranted.

Performance Requirements

The Transaction Service is expected to be implemented on a wide range of hardware and software platforms ranging from desktop computers to massively parallel servers and in networks ranging in size from a single LAN to worldwide networks. To meet this wide range of requirements, consideration must be given to algorithms which scale, efficient communications, and the number and size of accesses to permanent storage. Much of this is implementation, and therefore not visible to the user of the service. Nevertheless, the expected performance of the Transaction Service was compared to its procedural equivalent, the X/Open DTP model in the following areas:

- The number of network messages required.
- The number of disk accesses required.
- The amount of data logged.

The objective of the specification was to achieve parity with the X/Open model for equivalent function, where technically feasible.

10.2 Service Architecture

Figure 10-2 illustrates the major components and interfaces defined by the Transaction Service. The transaction originator is an arbitrary program that begins a transaction. The recoverable server implements an object with recoverable state that is invoked within the scope of the transaction, either directly by the transaction originator or indirectly through one or more transactional objects.

The transaction originator creates a transaction using a *TransactionFactory*; a *Control* is returned that provides access to a *Terminator* and a *Coordinator*. The transaction originator uses the *Terminator* to commit or rollback the transaction. The *Coordinator* is made available to recoverable servers, either explicitly or implicitly (by implicitly propagating a transaction context with a request). A recoverable server registers a *Resource* with the *Coordinator*. The *Resource* implements the two-phase commit protocol which is driven by the Transaction Service. A recoverable server may register a *Synchronization* with the *Coordinator*. The *Synchronization* implements a dependent object protocol driven by the Transaction Service. A recoverable server can also register a specialized resource called a *SubtransactionAwareResource* to track the completion of subtransactions. A *Resource* uses a *RecoveryCoordinator* in certain failure cases to determine the outcome of the transaction and to coordinate the recovery process with the Transaction Service.

To simplify coding, most applications use the *Current* pseudo object, which provides access to an implicit per-thread transaction context.

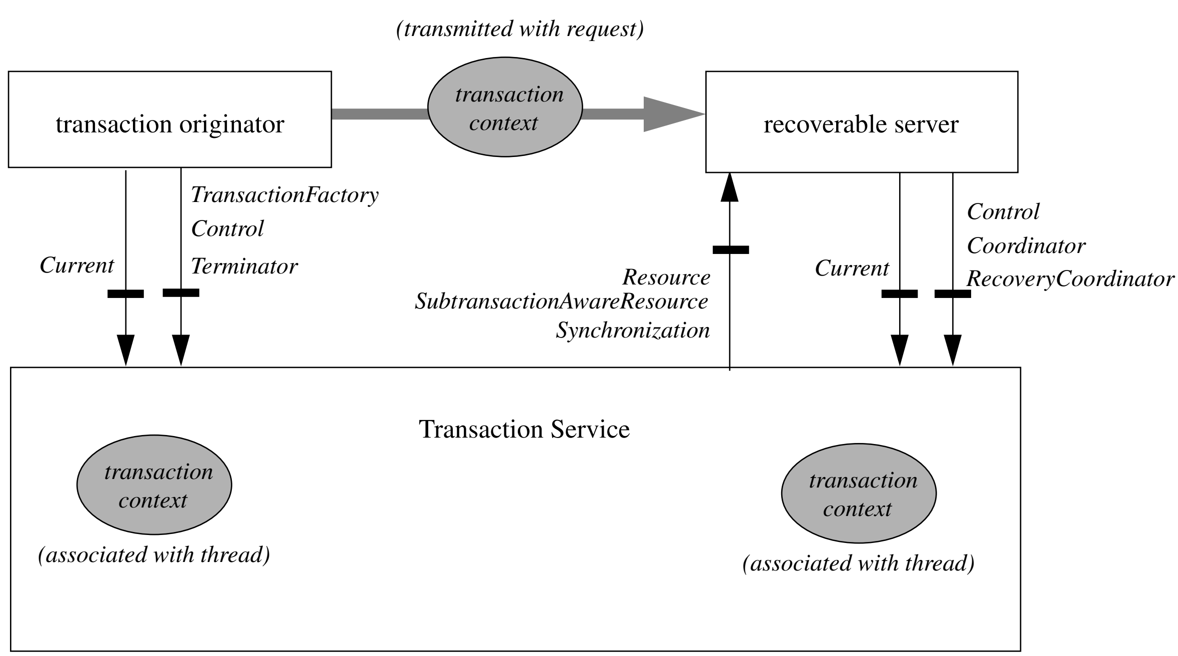


Figure 10-2 Major Components and Interfaces of the Transaction Service

10.2.1 Typical Usage

A typical transaction originator uses the *Current* object to begin a transaction, which becomes associated with the transaction originator's thread.

The transaction originator then issues requests. Some of these requests involve transactional objects. When a request is issued to a transactional object, the transaction context associated with the invoking thread is automatically propagated to the thread executing the method of the target object. No explicit operation parameter or context declaration is required to transmit the transaction context. Propagation of the transaction context can extend to multiple levels if a transactional object issues a request to a transactional object.

Using the *Current* object, the transactional object can unilaterally rollback the transaction and can inquire about the current state of the transaction. Using the *Current* object, the transactional object also can obtain a *Coordinator* for the current transaction. Using the *Coordinator*, a transactional object can determine the relationship between two transactions, to implement isolation among multiple transactions.

Some transactional objects are also recoverable objects. A recoverable object has persistent data that must be managed as part of the transaction. A recoverable object uses the *Coordinator* to register a *Resource* object as a participant in the transaction. The resource represents the recoverable object's participation in the transaction; each resource is implicitly associated with a single transaction. The *Coordinator* uses the resource to perform the two-phase commit protocol on the recoverable object's data.

After the computations involved in the transaction have been completed, the transaction originator uses the *Current* object to request that the changes be committed. The Transaction Service commits the transaction using a two-phase commit protocol wherein a series of requests are issued to the registered resources.

10.2.2 Transaction Context

The transaction context associated with a thread is either null (indicating that the thread has no associated transaction) or it refers to a specific transaction. It is permitted for multiple threads to be associated with the same transaction at the same time.

When a thread in an object server is used by an object adapter to perform a request on a transactional object, the object adapter initializes the transaction context associated with that thread by effectively copying the transaction context of the thread that issued the request. An implementation of the Transaction Service may restrict the capabilities of the new transaction context. For example, an implementation of the Transaction Service might not permit the object server thread to request commitment of the transaction.

The object adapter is not required to initialize the transaction context of every request handler. It is required to initialize the transaction context only if the interface supported by the target object is derived from the *TransactionalObject* interface. Otherwise, the initial transaction context of the thread is undefined.

When a thread retrieves the response to a deferred synchronous request, an exception may be raised if the thread is no longer associated with the transaction that it was associated with when the deferred synchronous request was issued. (See "Exceptions" on page 10-16, subsection "WRONG_TRANSACTION Exception" for a more precise definition.)

When nested transactions are used, the transaction context remembers the stack of nested transactions started within a particular execution environment (e.g., process) so that when a subtransaction ends, the transaction context of the thread is restored to the context in effect when the subtransaction was begun. When the context is transferred between execution environments, the received context refers only to one particular transaction, not a stack of transactions.

10.2.3 Context Management

The Transaction Service supports management and propagation of transaction context using objects provided by the Transaction Service. Using this approach, the transaction originator issues a request to a *TransactionFactory* to begin a new top-level transaction. The factory returns a *Control* object specific to the new transaction that allows an application to terminate the transaction or to become a participant in the transaction (by registering a *Resource*). An application can propagate a transaction context by passing the *Control* as an explicit request parameter.

The *Control* does not directly support management of the transaction. Instead, it supports operations that return two other objects, a *Terminator* and a *Coordinator*. The *Terminator* is used to commit or rollback the transaction. The *Coordinator* is used to enable transactional objects to participate in the transaction. These two objects can be propagated independently, allowing finer granularity control over propagation.

An implementation of the Transaction Service may restrict the ability for some or all of these objects to be transmitted to or used in other execution environments, to enable it to guarantee transaction integrity.

An application can also use the *Current* object operations get_control, suspend, and resume to obtain or change the implicit transaction context associated with its thread.

When nested transactions are used, a *Control* can include a stack of nested transactions begun in the same execution environment. When a *Control* is transferred between execution environments, the received *Control* refers only to one particular transaction, not a stack of transactions.

10.2.4 Datatypes

The CosTransactions module defines the following datatypes:

```
enum Status {
        StatusActive,
        StatusMarkedRollback,
        StatusPrepared,
        StatusCommitted,
        StatusRolledBack,
        StatusUnknown,
        StatusNoTransaction,
        StatusPreparing,
        StatusCommitting,
        StatusRollingBack
};

enum Vote {
        VoteCommit,
        VoteRollback,
        VoteReadOnly
};
```

10.2.5 Structures

The CosTransactions module defines the following structures:

```
struct otid_t {
        long formatID; /*format identifier. 0 is OSI TP */
        long bqual_length;
        sequence <octet> tid;
};
struct TransIdentity {
        Coordinator coord;
        Terminator term;
        otid_t otid;
};
struct PropagationContext {
        unsigned long timeout;
        TransIdentity current;
        sequence <TransIdentity> parents;
        any implementation_specific_data;
};
```

10.2.6 Exceptions

Standard Exceptions

The *CosTransactions* module adds new standard exceptions to CORBA for TRANSACTION_REQUIRED, TRANSACTION_ROLLEDBACK, and INVALID_TRANSACTION. These exceptions are defined in Chapter 3, Section 3.15 of the *Common Object Request Broker: Architecture and Specification.*

Heuristic Exceptions

A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service. Heuristic decisions are normally made only in unusual circumstances, such as communication failures, that prevent normal processing. When a heuristic decision is taken, there is a risk that the decision will differ from the consensus outcome, resulting in a loss of data integrity.

The CosTransactions module defines the following exceptions for reporting incorrect heuristic decisions or the possibility of incorrect heuristic decisions:

```
exception HeuristicRollback {};
exception HeuristicCommit {};
exception HeuristicMixed {};
exception HeuristicHazard {};
```

HeuristicRollback Exception

The commit operation on *Resource* raises the HeuristicRollback exception to report that a heuristic decision was made and that all relevant updates have been rolled back.

HeuristicCommit Exception

The rollback operation on *Resource* raises the HeuristicCommit exception to report that a heuristic decision was made and that all relevant updates have been committed.

HeuristicMixed Exception

A request raises the HeuristicMixed exception to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

HeuristicHazard Exception

A request raises the HeuristicHazard exception to report that a heuristic decision may have been made, the disposition of all relevant updates is not known, and for those updates whose disposition is known, either all have been committed or all have been rolled back. (In other words, the HeuristicMixed exception takes priority over the HeuristicHazard exception.)

WRONG_TRANSACTION Exception

The *CosTransactions* module adds the WRONG_TRANSACTION exception that can be raised by the ORB when returning the response to a deferred synchronous request. This exception is defined in Chapter 4 of the *Common Object Request Broker: Architecture and Specification.*

Other Exceptions

The CosTransactions module defines the following additional exceptions:

```
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};
```

These exceptions are described below along with the operations that raise them.

10.3 Transaction Service Interfaces

The interfaces defined by the Transaction Service reside in the CosTransactions module. (OMG IDL for the CosTransactions module is shown in “The CosTransactions Module” on page 10-69.) The interfaces for the Transaction Service are as follows:

- *Current*
- *TransactionFactory*
- *Terminator*
- *Coordinator*
- *RecoveryCoordinator*
- *Resource*
- *Synchronization*
- *Subtransaction Aware Resource*
- *Transactional Object*

No operations are defined in these interfaces for destroying objects. No application actions are required to destroy objects that support the Transaction Service because the Transaction Service destroys its own objects when they are no longer needed.

10.3.1 Current Interface

The *Current* interface defines operations that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. The *Current* interface also defines operations that simplify the use of the Transaction Service for most applications. These operations can be used to begin and end transactions and to obtain information about the current transaction.

The *Current* interface is designed to be supported by a pseudo object whose behavior depends upon and may alter the transaction context associated with the invoking thread. It may be shared with other object services (e.g., security) and is obtained by using a resolve initial references(“TransactionCurrent”) operation on the CORBA::ORB interface. *Current* supports the following operations:

```
interface Current : CORBA::Current {
	void begin()
		raises(SubtransactionsUnavailable);
	void commit(in boolean report_heuristics)
		raises(
			NoTransaction,
			HeuristicMixed,
			HeuristicHazard
		);
	void rollback()
		raises(NoTransaction);
	void rollback_only()
		raises(NoTransaction);

	Status get_status();
	string get_transaction_name();
	void set_timeout(in unsigned long seconds);

	Control get_control();
	Control suspend();
	void resume(in Control which)
		raises(InvalidControl);
};
```

Note – In order to pass the transaction from one thread to another, a program should not use the Current object. It should pass the Control object to the other thread.

begin

A new transaction is created. The transaction context of the client thread is modified so that the thread is associated with the new transaction. If the client thread is currently associated with a transaction, the new transaction is a subtransaction of that transaction. Otherwise, the new transaction is a top-level transaction.

The SubtransactionsUnavailable exception is raised if the client thread already has an associated transaction and the Transaction Service implementation does not support nested transactions.

commit

If there is no transaction associated with the client thread, the NoTransaction exception is raised. If the client thread does not have permission to commit the transaction, the standard exception NO_PERMISSION is raised. (The commit operation may be restricted to the transaction originator in some implementations.)

Otherwise, the transaction associated with the client thread is completed. The effect of this request is equivalent to performing the commit operation on the corresponding *Terminator* object (see "Terminator Interface" on page 10-23); see "Terminator Interface and "Exceptions" on page 10-16 for a description of the exceptions that may be raised.

The client thread transaction context is modified as follows: If the transaction was begun by a thread (invoking begin) in the same execution environment, then the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to null.

rollback

If there is no transaction associated with the client thread, the NoTransaction exception is raised. If the client thread does not have permission to rollback the transaction, the standard exception NO_PERMISSION is raised. (The rollback operation may be restricted to the transaction originator in some implementations; however, the rollback_only operation, described below, is available to all transaction participants.)

Otherwise, the transaction associated with the client thread is rolled back. The effect of this request is equivalent to performing the rollback operation on the corresponding *Terminator* object (see "Terminator Interface" on page 10-23).

The client thread transaction context is modified as follows: If the transaction was begun by a thread (invoking begin) in the same execution environment, then the thread's transaction context is restored to its state prior to the begin request. Otherwise, the thread's transaction context is set to null.

rollback_only

If there is no transaction associated with the client thread, the NoTransaction exception is raised. Otherwise, the transaction associated with the client thread is modified so that the only possible outcome is to rollback the transaction. The effect of this request is equivalent to performing the rollback_only operation on the corresponding *Coordinator* object (see "Coordinator Interface" on page 10-24).

get_status

If there is no transaction associated with the client thread, the StatusNoTransaction value is returned. Otherwise, this operation returns the status of the transaction associated with the client thread. The effect of this request is equivalent to performing the get_status operation on the corresponding *Coordinator* object (see "Coordinator Interface" on page 10-24).

get_transaction_name

If there is no transaction associated with the client thread, an empty string is returned. Otherwise, this operation returns a printable string describing the transaction. The returned string is intended to support debugging. The effect of this request is equivalent to performing the get_transaction_name operation on the corresponding *Coordinator* object (see "Coordinator Interface" on page 10-24).

set_timeout

This operation modifies a state variable associated with the target object that affects the time-out period associated with top-level transactions created by subsequent invocations of the begin operation. If the parameter has a nonzero value *n*, then top-level transactions created by subsequent invocations of begin will be subject to being rolled back if they do not complete before *n* seconds after their creation. If the parameter is zero, then no application specified time-out is established.

get_control

If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, a *Control* object is returned that represents the transaction context currently associated with the client thread. This object can be given to the resume operation to reestablish this context in the same thread or a different thread. The scope within which this object is valid is implementation dependent; at a minimum, it must be usable by the client thread. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

suspend

If the client thread is not associated with a transaction, a null object reference is returned. Otherwise, an object is returned that represents the transaction context currently associated with the client thread. This object can be given to the resume operation to reestablish this context in the same thread or a different thread. The scope within which this object is valid is implementation dependent; at a minimum, it must be usable by the client thread. In addition, the client thread becomes associated with no transaction. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

resume

If the parameter is a null object reference, the client thread becomes associated with no transaction. Otherwise, if the parameter is valid in the current execution environment, the client thread becomes associated with that transaction (in place of any previous transaction). Otherwise, the InvalidControl exception is raised. See "Control Interface" on page 10-22 for a discussion of restrictions on the scope of a *Control*. This operation is not dependent on the state of the transaction; in particular, it does not raise the TRANSACTION_ROLLEDBACK exception.

10.3.2 TransactionFactory Interface

The *TransactionFactory* interface is provided to allow the transaction originator to begin a transaction. This interface defines two operations, create and recreate, which create a new representation of a top-level transaction. A *TransactionFactory* is located using the *FactoryFinder* interface of the life cycle service and not by the resolve_initial_reference operation on the *ORB* interface defined in "Example Object Adapters" in Chapter 2 of the *Common Object Request Broker: Architecture and Specification.*

```
interface TransactionFactory {
        Control create(in unsigned long time_out);
        Control recreate(in PropagationContext ctx);
};
```

create

A new top-level transaction is created and a *Control* object is returned. The *Control* object can be used to manage or to control participation in the new transaction. An implementation of the Transaction Service may restrict the ability for the *Control* object to be transmitted to or used in other execution environments; at a minimum, it can be used by the client thread.

If the parameter has a nonzero value *n*, then the new transaction will be subject to being rolled back if it does not complete before *n* seconds have elapsed. If the parameter is zero, then no application specified time-out is established.

recreate

A new representation is created for an existing transaction defined by the *PropagationContext* and a *Control* object is returned. The *Control* object can be used to manage or to control participation in the transaction. An implementation of the Transaction Service which supports interposition (see "ORB/TS Implementation Considerations" on page 10-60) uses recreate to create a new representation of the transaction being imported, subordinate to the representation in ctx. The recreate operation can also be used to import a transaction which originated outside of the Transaction Service.

10.3.3 Control Interface

The *Control* interface allows a program to explicitly manage or propagate a transaction context. An object supporting the *Control* interface is implicitly associated with one specific transaction.

```
interface Control {
        Terminator get_terminator()
                raises(Unavailable);
        Coordinator get_coordinator()
                raises(Unavailable);
};
```

The *Control* interface defines two operations, get_terminator and get_coordinator. The get_terminator operation returns a *Terminator* object, which supports operations to end the transaction. The get_coordinator operation returns a *Coordinator* object, which supports operations needed by resources to participate in the transaction. The two objects support operations that are typically performed by different parties. Providing two objects allows each set of operations to be made available only to the parties that require those operations.

A *Control* object for a transaction is obtained using the operations defined by the *TransactionFactory* interface or the create_subtransaction operation defined by the *Coordinator* interface. It is possible to obtain a *Control* object for the current transaction (associated with a thread) using the get_control or suspend operations defined by the *Current* interface (see “Current Interface” on page 10-18). (These two operations return a null object reference if there is no current transaction.)

An implementation of the Transaction Service may restrict the ability for the *Control* object to be transmitted to or used in other execution environments; at a minimum, it can be used within a single thread.

get_terminator

An object is returned that supports the *Terminator* interface. The object can be used to rollback or commit the transaction associated with the *Control*. The Unavailable exception may be raised if the *Control* cannot provide the requested object. An implementation of the Transaction Service may restrict the ability for the *Terminator* object to be transmitted to or used in other execution environments; at a minimum, it can be used within the client thread.

get_coordinator

An object is returned that supports the *Coordinator* interface. The object can be used to register resources for the transaction associated with the *Control*. The Unavailable exception may be raised if the *Control* cannot provide the requested object. An implementation of the Transaction Service may restrict the ability for the *Coordinator* object to be transmitted to or used in other execution environments; at a minimum, it can be used within the client thread.

10.3.4 Terminator Interface

The *Terminator* interface supports operations to commit or rollback a transaction. Typically, these operations are used by the transaction originator.

```
interface Terminator {
	void commit(in boolean report_heuristics)
		raises(
			HeuristicMixed,
			HeuristicHazard
		);
	void rollback();
};
```

An implementation of the Transaction Service may restrict the scope in which a *Terminator* can be used; at a minimum, it can be used within a single thread.

commit

If the transaction has not been marked rollback only, and all of the participants in the transaction agree to commit, the transaction is committed and the operation terminates normally. Otherwise, the transaction is rolled back (as described below) and the TRANSACTION_ROLLEDBACK standard exception is raised.

If the report_heuristics parameter is true, the Transaction Service will report inconsistent or possibly inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions (defined in "Exceptions" on page 10-16). A Transaction Service implementation may optionally use the Event Service to report heuristic decisions.

The commit operation may rollback the transaction if there are subtransactions of the transaction that have not themselves been committed or rolled back or if there are existing or potential activities associated with the transaction that have not completed. The nature and extent of such error checking is implementation-dependent.

When a top-level transaction is committed, all changes to recoverable objects made in the scope of this transaction are made permanent and visible to other transactions or clients. When a subtransaction is committed, the changes are made visible to other related transactions as appropriate to the degree of isolation enforced by the resources.

rollback

The transaction is rolled back.

When a transaction is rolled back, all changes to recoverable objects made in the scope of this transaction (including changes made by descendant transactions) are rolled back. All resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the resources.

10.3.5 Coordinator Interface

The *Coordinator* interface provides operations that are used by participants in a transaction. These participants are typically either recoverable objects or agents of recoverable objects, such as subordinate coordinators. Each object supporting the *Coordinator* interface is implicitly associated with a single transaction.

```
interface Coordinator {

        Status get_status();
        Status get_parent_status();
        Status get_top_level_status();

        boolean is_same_transaction(in Coordinator tc);
        boolean is_related_transaction(in Coordinator tc);
        boolean is_ancestor_transaction(in Coordinator tc);
        boolean is_descendant_transaction(in Coordinator tc);
        boolean is_top_level_transaction();

        unsigned long hash_transaction();
        unsigned long hash_top_level_tran();

        RecoveryCoordinator register_resource(in Resource r)
                raises(Inactive);

        void register_synchronization (in Synchronization sync)
                raises(Inactive, SynchronizationUnavailable);

};
```

.

```
        void register_subtran_aware(in SubtransactionAwareResource r)
                raises(Inactive, NotSubtransaction);

        void rollback_only()
                raises(Inactive);

        string get_transaction_name();

        Control create_subtransaction()
                raises(SubtransactionsUnavailable, Inactive);

        PropagationContext get_txcontext ()
                raises(Unavailable);
};
```

An implementation of the Transaction Service may restrict the scope in which a *Coordinator* can be used; at a minimum, it can be used within a single thread.

get_status

This operation returns the status of the transaction associated with the target object:

- StatusActive - A transaction is associated with the target object and it is in the active state. An implementation returns this status after a transaction has been started and prior to a coordinator issuing any prepares unless it has been marked for rollback.
- StatusMarkedRollback - A transaction is associated with the target object and has been marked for rollback, perhaps as the result of a rollback_only operation.
- StatusPrepared - A transaction is associated with the target object and has been prepared (i.e., all subordinates have responded VoteCommit). The target object may be waiting for a superior's instructions as to how to proceed.
- StatusCommitted - A transaction is associated with the target object and it has completed commitment. It is likely that heuristics exists; otherwise, the transaction would have been destroyed and StatusNoTransaction returned.
- StatusRolledBack - A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exists, otherwise the transaction would have been destroyed and StatusNoTransaction returned.
- StatusUnknown - A transaction is associated with the target object, but the Transaction Service cannot determine its current status. This is a transient condition, and a subsequent invocation will ultimately return a different status.
- StatusNoTransaction - No transaction is currently associated with the target object. This will occur after a transaction has completed.
- StatusPreparing - A transaction is associated with the target object and it is the process of preparing. An implementation returns this status if it has started preparing, but has not yet completed the process, probably because it is waiting for responses to prepare from one or more resources.
- StatusCommitting - A transaction is associated with the target object and is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process, probably because it is waiting for responses from one or more resources.
- StatusRollingBack - A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process, probably because it is waiting for responses from one or more resources.

get_parent_status

If the transaction associated with the target object is a top-level transaction, then this operation is equivalent to the get_status operation. Otherwise, this operation returns the status of the parent of the transaction associated with the target object.

get_top_level_status

This operation returns the status of the top-level ancestor of the transaction associated with the target object. If the transaction is a top-level transaction, then this operation is equivalent to the get_status operation.

is_same_transaction

This operation returns true if, and only if, the target object and the parameter object both refer to the same transaction.

is_ancestor_transaction

This operation returns true if, and only if, the transaction associated with the target object is an ancestor of the transaction associated with the parameter object. A transaction T1 is an ancestor of a transaction T2 if and only if T1 is the same as T2 or T1 is an ancestor of the parent of T2.

is_descendant_transaction

This operation returns true if, and only if, the transaction associated with the target object is a descendant of the transaction associated with the parameter object. A transaction T1 is a descendant of a transaction T2 if, and only if, T2 is an ancestor of T1 (see above).

is_related_transaction

This operation returns true if, and only if, the transaction associated with the target object is related to the transaction associated with the parameter object. A transaction T1 is related to a transaction T2 if, and only if, there exists a transaction T3 such that T3 is an ancestor of T1 and T3 is an ancestor of T2.

is_top_level_transaction

This operation returns true if, and only if, the transaction associated with the target object is a top-level transaction. A transaction is a top-level transaction if it has no parent.

hash_transaction

This operation returns a hash code for the transaction associated with the target object. Each transaction has a single hash code. Hash codes for transactions should be uniformly distributed.

hash_top_level_tran

This operation returns the hash code for the top-level ancestor of the transaction associated with the target object. This operation is equivalent to the hash_transaction operation when the transaction associated with the target object is a top-level transaction.

register_resource

This operation registers the specified resource as a participant in the transaction associated with the target object. When the transaction is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction. These requests are described in the description of the *Resource* interface. The Inactive exception is raised if the transaction has already been prepared. The standard exception TRANSACTION_ROLLEDBACK may be raised if the transaction has been marked rollback only.

If the resource is a subtransaction aware resource (it supports the *SubtransactionAwareResource* interface) and the transaction associated with the target object is a subtransaction, then this operation registers the specified resource with the subtransaction and indirectly with the top-level transaction when the subtransaction's ancestors have completed. Otherwise, the resource is registered as a participant in the current transaction. If the current transaction is a subtransaction, the resource will not receive prepare or commit requests until the top-level ancestor terminates.

This operation returns a *RecoveryCoordinator* that can be used by this resource during recovery.

register_synchronization

This operation registers the specified *Synchronization* object such that it will be notified to perform necessary processing prior to prepare being driven to resources registered with this *Coordinator*. These requests are described in the description of the *Synchronization* interface. The Inactive exception is raised if the transaction has already been prepared. The SynchronizationUnavailable exception is raised if the *Coordinator* does not support synchronization. The standard exception TRANSACTION_ROLLEDBACK may be raised if the transaction has been marked rollback only.

register_subtran_aware

This operation registers the specified subtransaction aware resource such that it will be notified when the subtransaction has committed or rolled back. These requests are described in the description of the *SubtransactionAwareResource* interface.

Note that this operation registers the specified resource only with the subtransaction. This operation cannot be used to register the resource as a participant in the transaction.

The NotSubtransaction exception is raised if the current transaction is not a subtransaction. The Inactive exception is raised if the subtransaction (or any ancestor) has already been terminated. The standard exception TRANSACTION_ROLLEDBACK may be raised if the subtransaction (or any ancestor) has been marked rollback only.

rollback_only

The transaction associated with the target object is modified so that the only possible outcome is to rollback the transaction. The Inactive exception is raised if the transaction has already been prepared.

get_transaction_name

This operation returns a printable string describing the transaction associated with the target object. The returned string is intended to support debugging.

create_subtransaction

A new subtransaction is created whose parent is the transaction associated with the target object. The Inactive exception is raised if the target transaction has already been prepared. An implementation of the Transaction Service is not required to support nested transactions. If nested transactions are not supported, the exception SubtransactionsUnavailable is raised.

The create_subtransaction operation returns a *Control* object, which enables the subtransaction to be terminated and allows recoverable objects to participate in the subtransaction. An implementation of the Transaction Service may restrict the ability for the *Control* object to be transmitted to or used in other execution environments.

get_txcontext

The get_txcontext operation returns a *PropagationContext* object, which is used by one Transaction Service domain to export the current transaction to a new Transaction Service domain. An implementation of the Transaction Service may also use the *PropagationContext* to assist in the implementation of the is_same_transaction operation when the input *Coordinator* has been generated by a different Transaction Service implementation.

The Unavailable exception is raised if the Transaction Service implementation chooses to restrict the availability of the *PropagationContext*.

10.3.6 Recovery Coordinator Interface

A recoverable object uses a *RecoveryCoordinator* to drive the recovery process in certain situations. The object reference for an object supporting the *RecoveryCoordinator* interface, as returned by the register_resource operation, is implicitly associated with a single resource registration request and may only be used by that resource.

```
interface RecoveryCoordinator {
        Status replay_completion(in Resource r)
                raises(NotPrepared);
};
```

replay_completion

This operation can be invoked at any time after the associated resource has been prepared. The *Resource* must be passed as the parameter. Performing this operation provides a hint to the *Coordinator* that the commit or rollback operations have not been performed on the resource. This hint may be required in certain failure cases. This non-blocking operation returns the current status of the transaction. The NotPrepared exception is raised if the resource has not been prepared.

10.3.7 Resource Interface

The Transaction Service uses a two-phase commitment protocol to complete a top-level transaction with each registered resource. The *Resource* interface defines the operations invoked by the transaction service on each resource. Each object supporting the *Resource* interface is implicitly associated with a single top-level transaction. Note that in the case of failure, the completion sequence will continue after the failure is repaired. A resource should be prepared to receive duplicate requests for the commit or rollback operation and to respond consistently.

```
interface Resource {
        Vote prepare()
                raises(
                        HeuristicMixed,
                        HeuristicHazard
                );
        void rollback()
                raises(
                        HeuristicCommit,
                        HeuristicMixed,
                        HeuristicHazard
                );
        void commit()
                raises(
                        NotPrepared,
                        HeuristicRollback,
                        HeuristicMixed,
                        HeuristicHazard
                );
        void commit_one_phase()
                raises(
                        HeuristicHazard
                );
        void forget();
};
```

prepare

This operation is invoked to begin the two-phase commit protocol on the resource. The resource can respond in several ways, represented by the Vote result.

If no persistent data associated with the resource has been modified by the transaction, the resource can return VoteReadOnly. After receiving this response, the Transaction Service is not required to perform any additional operations on this resource. Furthermore, the resource can forget all knowledge of the transaction.

If the resource is able to write (or has already written) all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction, it can return VoteCommit. After receiving this response, the Transaction Service is required to eventually perform either the commit or the rollback operation on this object. To support recovery, the resource should store the *RecoveryCoordinator* object reference in stable storage.

The resource can return VoteRollback under any circumstances, including not having any knowledge about the transaction (which might happen after a crash). If this response is returned, the transaction must be rolled back. Furthermore, the Transaction Service is not required to perform any additional operations on this resource. After returning this response, the resource can forget all knowledge of the transaction.

The resource reports inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions (described in “Exceptions” on page 10-16). Heuristic outcomes occur when a resource acts as a sub-coordinator and at least one of its resources takes a heuristic decision after a VoteCommit return.

rollback

If necessary, the resource should rollback all changes made as part of the transaction. If the resource has forgotten the transaction, it should do nothing.

The heuristic outcome exceptions (described in “Exceptions” on page 10-16) are used to report heuristic decisions related to the resource. If a heuristic outcome exception is raised, the resource must remember this outcome until the forget operation is performed so that it can return the same outcome in case rollback is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

commit

If necessary, the resource should commit all changes made as part of the transaction. If the resource has forgotten the transaction, it should do nothing.

The heuristic outcome exceptions (described in “Exceptions” on page 10-16) are used to report heuristic decisions related to the resource. If a heuristic outcome exception is raised, the resource must remember this outcome until the forget operation is performed so that it can return the same outcome in case commit is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

The NotPrepared exception is raised if the commit operation is performed without first performing the prepare operation.

commit_one_phase

If possible, the resource should commit all changes made as part of the transaction. If it cannot, it should raise the TRANSACTION_ROLLEDBACK standard exception.

If a failure occurs during commit_one_phase, it must be retried when the failure is repaired. Since their can only be a single resource, the HeuristicHazard exception is used to report heuristic decisions related to that resource. If a heuristic exception is raised, the resource must remember this outcome until the forget operation is performed so that it can return the same outcome in case commit_one_phase is performed again. Otherwise, the resource can immediately forget all knowledge of the transaction.

forget

This operation is performed only if the resource raised a heuristic outcome exception to rollback, commit, or commit_one_phase. Once the coordinator has determined that the heuristic situation has been addressed, it should issue forget on the resource. The resource can forget all knowledge of the transaction.

10.3.8 Synchronization Interface

The Transaction Service provides a synchronization protocol which enables an object with transient state data that relies on an X/Open XA conformant Resource Manager for ensuring that data is made persistent, to be notified before the start of the two-phase commitment protocol, and after its completion. An object with transient state data that relies on a *Resource* object for ensuring that data is made persistent can also make use of this protocol, provided that both objects are registered with the same *Coordinator*. Each object supporting the *Synchronization* interface is implicitly associated with a single top-level transaction.

```
interface Synchronization : TransactionalObject {
	void before_completion();
	void after_completion(in Status status);
};
```

before_completion

This operation is invoked prior to the start of the two-phase commit protocol within the coordinator the *Synchronization* has registered with. This operation will therefore be invoked prior to prepare being issued to *Resource* objects or X/Open Resource Managers registered with that same coordinator. The *Synchronization* object must ensure that any state data it has that needs to be made persistent is made available to the resource.

Only standard exceptions may be raised. Unless there is a defined recovery procedure for the exception raised, the transaction should be marked rollback only.

after_completion

This operation is invoked after all commit or rollback responses have been received by this coordinator. The current status of the transaction (as determined by a get_status on the *Coordinator*) is provided as input.

Only standard exceptions may be raised and they have no effect on the outcome of the commitment process.

10.3.9 Subtransaction Aware Resource Interface

Recoverable objects that implement nested transaction behavior may support a specialization of the *Resource* interface called the *SubtransactionAwareResource* interface. A recoverable object can be notified of the completion of a subtransaction by registering a specialized resource object that offers the *SubtransactionAwareResource* interface with the Transaction Service. This registration is done by using the register_resource or the register_subtran_aware operation of the current *Coordinator* object. A recoverable object generally uses the register_resource operation to register a resource that will participate in the completion of the top-level transaction and the register_subtran_aware operation to be notified of the completion of a subtransaction.

Certain recoverable objects may want a finer control over the registration in the completion of a subtransaction. These recoverable objects will use the register_resource operation to ensure participation in the completion of the top-level transaction and they will use the register_subtran_aware operation to be notified of the completion of a particular subtransaction. For example, a recoverable object can use the register_subtran_aware operation to establish a "committed with respect to" relationship between transactions; that is, the recoverable object wants to be informed when a particular subtransaction is committed and then perform certain operations on the transactions that depend on that transaction's completion. This technique could be used to implement lock inheritance, for example.

The Transaction Service uses the *SubtransactionAwareResource* interface on each *Resource* object registered with a subtransaction. Each object supporting this interface is implicitly associated with a single subtransaction.

```
interface SubtransactionAwareResource : Resource {
        void commit_subtransaction(in Coordinator parent);
        void rollback_subtransaction();
};
```

commit_subtransaction

This operation is invoked only if the resource has been registered with a subtransaction and the subtransaction has been committed. The *Resource* object is provided with a *Coordinator* that represents the parent transaction. This operation may raise a standard exception such as TRANSACTION_ROLLEDBACK.

Note that the results of a committed subtransaction are relative to the completion of its ancestor transactions, that is, these results can be undone if any ancestor transaction is rolled back.

rollback_subtransaction

This operation is invoked only if the resource has been registered with a subtransaction and notifies the resource that the subtransaction has rolled back.

10.3.10 TransactionalObject Interface

The *TransactionalObject* interface is used by an object to indicate that it is transactional. By supporting the *TransactionalObject* interface, an object indicates that it wants the transaction context associated with the client thread to be associated with all operations on its interface.

```
interface TransactionalObject {
};
```

The *TransactionalObject* interface defines no operations. It is simply a marker.

10.4 The User's View

The audience for this section is object and client implementers; it describes application use of the Transaction Service functions.

10.4.1 Application Programming Models

A client application program may use direct or indirect context management to manage a transaction.

- With indirect context management, an application uses the *Current* object provided by the Transaction Service, to associate the transaction context with the application thread of control.
- In direct context management, an application manipulates the *Control* object and the other objects associated with the transaction.

Propagation is the act of associating a client's transaction context with operations on a target object. An object may require transactions to be either explicitly or implicitly propagated on its operations.

Implicit propagation means that requests are implicitly associated with the client's transaction; they share the client's transaction context. It is transmitted implicitly to the objects, without direct client intervention. Implicit propagation depends on indirect context management, since it propagates the transaction context associated with the *Current* object. ***Explicit propagation*** means that an application propagates a transaction context by passing objects defined by the Transaction Service as explicit parameters.

An object that supports implicit propagation would not typically expect to receive any Transaction Service object as an explicit parameter.

A client may use one or both forms of context management, and may communicate with objects that use either method of transaction propagation.

This results in four ways in which client applications may communicate with transactional objects. They are described below.

Direct Context Management: Explicit Propagation

The client application directly accesses the *Control* object, and the other objects which describe the state of the transaction. To propagate the transaction to an object, the client must include the appropriate Transaction Service object as an explicit parameter of an operation.

Indirect Context Management: Implicit Propagation

The client application uses operations on the *Current* object to create and control its transactions. When it issues requests on transactional objects, the transaction context associated with the current thread is implicitly propagated to the object.

Indirect Context Management: Explicit Propagation

For an implicit model application to use explicit propagation, it can get access to the *Control* using the get_control operation on *Current*. It can then use a Transaction Service object as an explicit parameter to a transactional object. This is explicit propagation.

Direct Context Management: Implicit Propagation

A client that accesses the Transaction Service objects directly can use the resume operation on *Current* to set the implicit transaction context associated with its thread. This allows the client to invoke operations of an object that requires implicit propagation of the transaction context.

10.4.2 Interfaces

Table 10-1 Use of Transaction Service Functionality

Function	Used by	Context management Direct	Indirect[1]
Create a transaction	Transaction originator	TransactionFactory::create Control::get_terminator Control::get_coordinator	begin,set_timeout
Terminate a transaction	Transaction originator—*implicit* All—*explicit*	Terminator::commit Terminator::rollback	commit rollback
Rollback a transaction	Server	Terminator::rollback_only	rollback_only
Control propagation of transaction to a server	Server	Declaration of method parameter	TransactionalObject interface
Control by client of transaction propagation to a server	All	Request parameters	get_control suspend resume
Become a participant in a transaction	Recoverable Server	Coordinator::register_resource	Not applicable
Miscellaneous	All	Coordinator::get_status Coordinator::get_transaction_name Coordinator::is_same_transaction Coordinator::hash_transaction	get_status get_transaction_name Not applicable Not applicable

1. All Indirect context management operations are on the *Current* object interface

Note – For clarity, subtransaction operations are not shown.

10.4.3 Checked Transaction Behavior

Some Transaction Service implementations will enforce checked behavior for the transactions they support, to provide an extra level of transaction integrity. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. A checked Transaction Service guarantees that commit will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests.

There are many possible implementations of checking in a Transaction Service. One provides equivalent function to that provided by the request/response inter-process communication models defined by X/Open.

The X/Open Transaction Service model of checking is particularly important because it is widely implemented. It describes the transaction integrity guarantees provided by many existing transaction systems. These transaction systems will provide the same level of transaction integrity for object-based applications by providing a Transaction Service interface that implements the X/Open checks.

10.4.4 X/Open Checked Transactions

In X/Open, completion of the processing of a request means that the object has completed execution of its method and replied to the request.

The level of transaction integrity provided by a Transaction Service implementing the X/Open model of checking provides equivalent function to that provided by the XATMI and TxRPC interfaces defined by X/Open for transactional applications. X/Open DTP Transaction Managers are examples of transaction management functions that implement checked transaction behavior.

This implementation of checked behavior depends on implicit transaction propagation. When implicit propagation is used, the objects involved in a transaction at any given time may be represented as a tree, the request tree for the transaction. The beginner of the transaction is the root of the tree. Requests add nodes to the tree, replies remove the replying node from the tree. Synchronous requests, or the checks described below for deferred synchronous requests, ensure that the tree collapses to a single node before commit is issued.

If a transaction uses explicit propagation, the Transaction Service cannot know which objects are or will be involved in the transaction; that is, a request tree cannot be constructed or assured. Therefore, the use of explicit propagation is not permitted by a Transaction Service implementation that enforces X/Open-style checked behavior.

Applications that use synchronous requests implicitly exhibit checked behavior. For applications that use deferred synchronous requests, in a transaction where all clients and objects are in the domain of a checking Transaction Service, the Transaction Service can enforce this property by applying a reply check and a commit check.

The Transaction Service must also apply a resume check to ensure that the transaction is only resumed by application programs in the correct part of the request tree.

Reply Check

Before allowing an object to reply to a transactional request, a check is made to ensure that the object has received replies to all its deferred synchronous requests that propagated the transaction in the original request. If this condition is not met, an exception is raised and the transaction is marked as rollback-only, that is, it cannot be successfully committed.

A Transaction Service may check that a reply is issued within the context of the transaction associated with the request.

Commit Check

Before allowing commit to proceed, a check is made to ensure that:

1. The commit request for the transaction is being issued from the same execution environment that created the transaction.

2. The client issuing commit has received replies to all the deferred synchronous requests it made that caused the propagation of the transaction.

Resume Check

Before allowing a client or object to associate a transaction context with its thread of control, a check is made to ensure that this transaction context was previously associated with the execution environment of the thread. This would be true if the thread either created the transaction or received it in a transactional operation.

10.4.5 Implementing a Transactional Client: Heuristic Completions

The commit operation takes the boolean report_heuristics as input. If the report_heuristics argument is false, commit can complete as soon as the root coordinator has made its decision to commit or rollback the transaction. The application is not required to wait for the coordinator to complete the commit protocol by informing all the participants of the outcome of the transaction. This can significantly reduce the elapsed time for the commit operation, especially where participant *Resource* objects are located on remote network nodes. However, no heuristic conditions can be reported to the application in this case.

Using the report_heuristics option guarantees that the commit operation will not complete until the coordinator has completed the commit protocol with all resources involved in the transaction. This guarantees that the application will be informed of any non-atomic outcomes of the transaction via the HeuristicMixed or HeuristicHazard exceptions, but increases the application-perceived elapsed time for the commit operation.

10.4.6 Implementing a Recoverable Server

A Recoverable Server includes at least one recoverable object and one *Resource* object. The responsibilities of each of these objects are explained in the following sections.

Recoverable Object

The responsibilities of the recoverable object are to implement the object's operations, and to register a *Resource* object with the *Coordinator* so commitment of the recoverable object's resources, including any necessary recovery, can be completed.

The *Resource* object identifies the involvement of the recoverable object in a particular transaction. This means a *Resource* object may only be registered in one transaction at a time. A different *Resource* object must be registered for each transaction in which a recoverable object is concurrently involved.

A recoverable object may receive multiple requests within the scope of a single transaction. It only needs to register its involvement in the transaction once. The is_same_transaction operation allows the recoverable object to determine if the transaction associated with the request is one in which the recoverable object is already registered.

The hash_transaction operations allow the recoverable object to reduce the number of transaction comparisons it has to make. All coordinators for the same transaction return the same hash code. The is_same_transaction operation need only be done on coordinators which have the same hash code as the coordinator of the current request.

Resource Object

The responsibilities of a *Resource* object are to participate in the completion of the transaction, to update the Recoverable Server's resources in accordance with the transaction outcome, and ensure termination of the transaction, including across failures. The protocols that the *Resource* object must follow are described in "Transaction Service Protocols" on page 10-49.

Reliable Servers

A Reliable Server is a special case of a Recoverable Server. A Reliable Server can use the same interface as a Recoverable Server to ensure application integrity for objects that do not have recoverable state. In the case of a Reliable Server, the recoverable object can register a *Resource* object that replies VoteReadOnly to prepare if its integrity constraints are satisfied (e.g., all debits have a corresponding credit), or replies VoteRollback if there is a problem. This approach allows the server to apply integrity constraints which apply to the transaction as a whole, rather than to individual requests to the server.

10.4.7 Application Portability

This section considers application portability across the broadest range of Transaction Service implementations.

Flat Transactions

There is one optional function of the Transaction Service, support for nested transactions. For an application to be portable across all implementations of the Transaction Service, it should be designed to use the flat transaction model. The Transaction Service specification treats flat transactions as top-level nested transactions.

X/Open Checked Transactions

Transaction Service implementations may implement checked or unchecked behavior. The transaction integrity checks implemented by a Transaction Service need not be the same as those defined by X/Open. However, many existing transaction management systems have implemented the X/Open model of interprocess communication, and will implement a checked Transaction Service that provides the same guarantee of transaction integrity.

Applications written to conform to the transaction integrity constraints of X/Open will be portable across all implementations of an X/Open checked Transaction Service, as well as all Transaction Service implementations which support unchecked behavior.

10.4.8 Distributed Transactions

The Transaction Service can be implemented by multiple components located across a network. The different components can be based on the same or on different implementations of the Transaction Service.

A single transaction can involve clients and objects supported by more than one instance of the Transaction Service. The number of Transaction Service instances involved in the transaction is not visible to the application implementer. There is no change in the function provided.

10.4.9 Applications Using Both Checked and Unchecked Services

A single transaction can include objects supported by both checked and unchecked Transaction Service implementations. Checked transaction behavior cannot be applied to the transaction as a whole.

It is possible to provide useful, limited forms of checked behavior for those subsets of the transaction's resources in the domain of a checked Transaction Service.

- First, a transactional or recoverable object, whose resources are managed by a checked Transaction Service, may be accessed by unchecked clients. The checked Transaction Service can ensure, by registering itself in the transaction, that the transaction will not commit before all the integrity constraints associated with the request have been satisfied.
- Second, an application whose resources are managed by a checked Transaction Service may act as a client of unchecked objects, and preserve its checked semantics.

10.4.10 Examples

Note – All the examples are written in pseudo code based on C++. In particular they do not include implicit parameters such as the ORB::Environment, which should appear in all requests. Also, they do not handle the exceptions that might be returned with each request.

A Transaction Originator: Indirect and Implicit

In the code fragments below, a transaction originator uses indirect context management and implicit transaction propagation; txn_crt is an example of an object supporting the *Current* interface; the client uses the begin operation to start the transaction which becomes implicitly associated with the originator's thread of control:

```
...
txn_crt.begin();
// should test the exceptions that might be raised
...
// the client issues requests, some of which involve
// transactional objects;
BankAccount1->makeDeposit(deposit);
...
```

The program commits the transaction associated with the client thread. The report_heuristics argument is set to false so no report will be made by the Transaction Service about possible heuristic decisions.

```
....
txn_crt.commit(false);
...
```

Transaction Originator: Direct and Explicit

In the following example, a transaction originator uses direct context management and explicit transaction propagation. The client uses a factory object supporting the CosTransactions::TransactionFactory interface to create a new transaction and uses the returned *Control* object to retrieve the *Terminator* and *Coordinator* objects.

```
...
CosTransactions::Control c;
CosTransactions::Terminator t;
CosTransactions::Coordinator co;

c = TFactory->create(0);
t = c->get_terminator();
...
```

The client issues requests, some of which involve transactional objects, in this case explicit propagation of the context is used. The *Control* object reference is passed as an explicit parameter of the request; it is declared in the OMG IDL of the interface.

```
...
transactional_object->do_operation(arg, c);
```

The transaction originator uses the *Terminator* object to commit the transaction; the report_heuristics argument is set to false: so no report will be made by the Transaction Service about possible heuristic decisions.

```
...
t->commit(false);
```

Example of a Recoverable Server

BankAccount1 is an object with internal resources. It inherits from both the *TransactionalObject* and the *Resource* interfaces:

```
interface BankAccount1:
       CosTransactions::TransactionalObject,CosTransactions::Resource
{
...
  void makeDeposit (in float amt);
...
};


class BankAccount1
{
public:
...
void makeDeposit(float amt);
...
}
```

Upon entering, the context of the transaction is implicitly associated with the object's thread. The pseudo object supporting the *Current* interface is used to retrieve the *Coordinator* object associated with the transaction.

```
void makeDeposit (float amt)
{
CosTransactions::Control c;
CosTransactions::Coordinator co;

c = txn_crt.get_control();
co = c->get_coordinator();
...
```

Before registering the *Resource,* the object must check whether it has already been registered for the same transaction. This is done using the hash_transaction and is_same_transaction operations on the current *Coordinator* to compare a list of saved coordinators representing currently active transactions. In this example, the object registers itself as a *Resource*. This requires the object to durably record its registration before issuing register_resource to handle potential failures and imposes the restriction that the object may only be involved in one transaction at a time.

If more parallelism is required, separate *Resource* objects can be registered for each transaction the object is involved in.

```
RecoveryCoordinator r;
r = co->register_resource (this);

// performs some transactional activity locally
balance = balance + f;
num_transactions++;
...
// end of transactional operation
};
```

Example of a Transactional Object

BankAccount2 is an object with external resources that inherits from the *TransactionalObject* interface:

```
interface BankAccount2: CosTransactions::TransactionalObject
{
...
  void makeDeposit(in float amt);
...
};


class BankAccount2
{
public:
...
void makeDeposit(float amt);
...
}
```

Upon entering, the context of the transaction is implicitly associated with the object's thread. The makeDeposit operation performs some transactional requests on external, recoverable servers. The objects res1 and res2 are recoverable objects. The current transaction context is implicitly propagated to these objects.

```
void makeDeposit(float amt)
{
        balance = res1->get_balance(amt);
        balance = balance + amt;
        res1->set_balance(balance);

        res2->increment_num_transactions();
} // end of transactional operation
```

10.4.11 Model Interoperability

The Transaction Service supports interoperability between Transaction Service applications using implicit context propagation and procedural applications using the X/Open DTP model. A single transaction management component may act as both the Transaction Service and an X/Open Transaction Manager.

Interoperability is provided in two ways:

- Importing transactions from the X/Open domain to the Transaction Service domain.
- Exporting transactions from the Transaction Service domain to the X/Open domain.

Importing Transactions

X/Open applications can access transactional objects. This means that an existing application, written to use X/Open interfaces, can be extended to invoke transactional operations. This causes the X/Open transaction to be imported into the domain of the Transaction Service.

The X/Open application may be a client or a server.

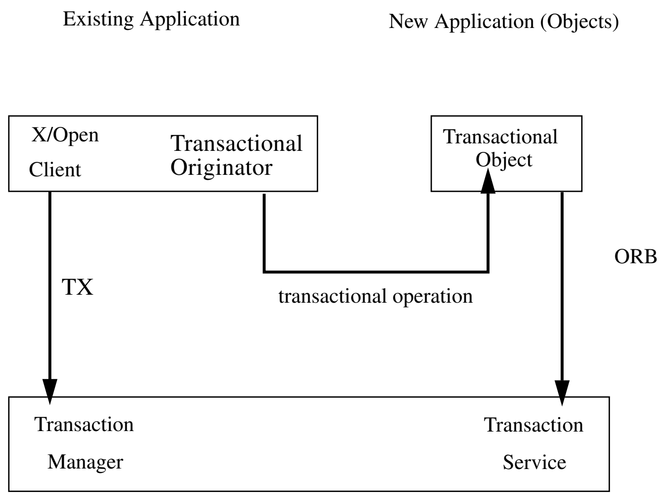


Figure 10-3 X/Open Client

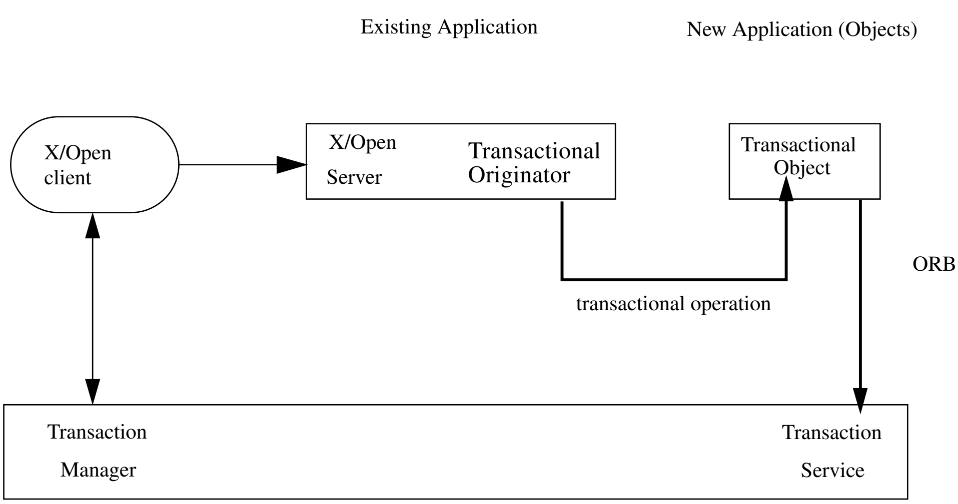


Figure 10-4 X/Open Server

Exporting Transactions

Transactional objects can use X/Open communications and resource manager interfaces, and include the resources managed by these components in a transaction managed by the Transaction Service. This causes the Transaction Service transaction to be exported into the domain of the X/Open transaction manager.

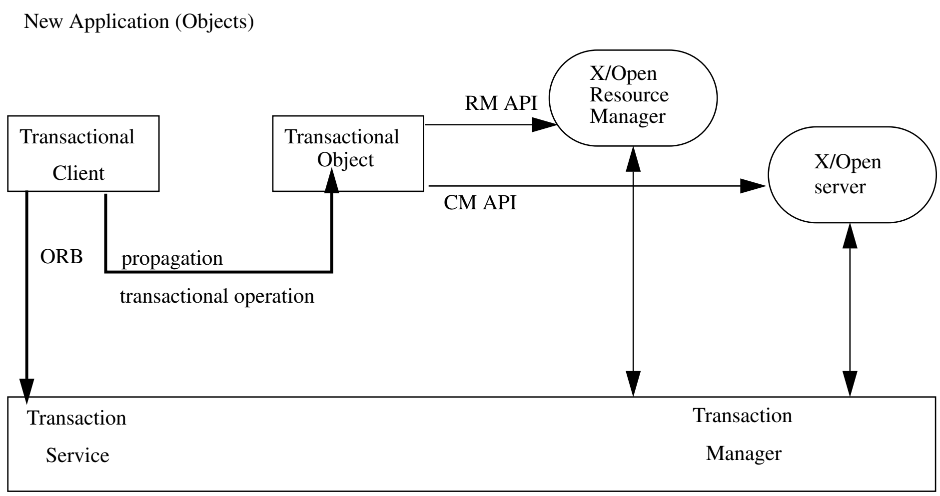


Figure 10-5 Sample Transaction Managed by the Transaction Service

Programming Rules

Model interoperability results in application programs that use both X/Open and Transaction Service interfaces.

A transaction originator may use the X/Open TX interface or the Transaction Service interfaces to create and terminate a transaction. Only one style may be used in one originator.

A single application may inherit a transaction with an application request either by using the X/Open server interfaces, or by being a transactional object.

Within a single transaction, an application program can be a client of both X/Open resource manager interfaces and transactional object interfaces.

An X/Open client or server may invoke operations of transactional objects. The X/Open transaction is imported into the Transaction Service domain using the recreate operation on *TransactionFactory*.

A transactional object with a *Current* object that associates a transaction context with a thread of control, can call X/Open Resource Managers. How requests to the X/Open Resource managers become associated with the transaction context of the *Current* object is implementation-dependent.

10.4.12 Failure Models

The Transaction Service provides atomic outcomes for transactions in the presence of application, system or communication failures. This section describes the behavior of application entities when failures occur. The protocols used to achieve this behavior are described in "Transaction Service Protocols" on page 10-49.

From the viewpoint of each user object role, two types of failure are relevant: a failure affecting the object itself (local failure) and a failure external to the object (external failure), such as failure of another object or failure in the communication with that object.

Transaction Originator

Local Failure

A failure of a transaction originator prior to the originator issuing commit will cause the transaction to be rolled back. A failure of the originator after issuing commit and before the outcome is reported may result in either commitment or rollback of the transaction depending on timing; in this case completion of the transaction takes place without regard to the failure of the originator.

External Failure

Any external failure affecting the transaction prior to the originator issuing commit will cause the transaction to be rolled back; the standard exception TRANSACTION_ROLLEDBACK will be raised in the originator when it issues commit.

A failure after commit and before the outcome has been reported will mean that the client may not be informed of the transaction outcome, depending on the nature of the failure, and the use of the report_heuristics option of commit. For example, the transaction outcome will not be reported to the client if communication between the client and the coordinator fails.

A client may use get_status on the *Coordinator* to determine the transaction outcome. However, this is not reliable because the status NoTransaction is ambiguous: it could mean that the transaction committed and has been forgotten, or that the transaction rolled back and has been forgotten.

If an originator needs to know the transaction outcome, including in the case of external failures, then either the originator's implementation must include a *Resource* object so that it will participate in the two-phase commit procedure (and any recovery), or the originator and coordinator must be located in the same failure domain (for example, the same execution environment).

Transactional Server

Local Failure

If the Transactional Server fails then optional checks by a Transaction Service implementation may cause the transaction to be rolled back; without such checks, whether the transaction is rolled back depends on whether the commit decision has already been made (this would be the case where an unchecked client invokes commit before receiving all replies from servers).

External Failure

Any external failure affecting the transaction during the execution of a Transactional Server will cause the transaction to be rolled back. If this occurs while the transactional object's method is executing, the failure has no effect on the execution of this method. The method may terminate normally, returning the reply to its client. Eventually the TRANSACTION_ROLLEDBACK exception will be returned to a client issuing commit.

Recoverable Server

Behavior of a recoverable server when failures occur is determined by the two phase commit protocol between the coordinator and the recoverable server's *Resource* object(s). This protocol, including the local and external failure models and the required behavior of the Resource, is described in "Transaction Service Protocols" on page 10-49.

10.5 The Implementers' View

This section contains three major categories of information.

1. "Transaction Service Protocols" on page 10-49 defines in more detail the protocols of the Transaction Service for ensuring atomicity of transactions, even in the presence of failure.

   This section is *not* a formal part of the specification but is provided to assist in building valid implementations of the specification. These protocols affect implementations of Recoverable Servers and the Transaction Service.

2. "ORB/TS Implementation Considerations" on page 10-60 provides additional information for implementers of ORBs and Transaction Services in those areas where cooperation between the two is necessary to realize the Transaction Service function.

   The following aspects of ORB and Transaction Service implementation are covered:
   - transaction propagation.
   - interoperation between different transaction service implementations.
   - ORB changes necessary to support portability of transaction service implementations.

3. "Model Interoperability" on page 10-67 describes how an implementation achieves interoperation between the Transaction Service and procedural transaction managers.

10.5.1 Transaction Service Protocols

The Transaction Service requires that certain protocols be followed to implement the atomicity property. These protocols affect the implementation of recoverable servers, (recoverable objects that register for participation in the two-phase commit process) and the coordinators that are created by a transaction factory. These responsibilities ensure the execution of the two-phase commit protocol and include maintaining state information in stable storage, so that transactions can be completed in case of failures.

General Principles

The first coordinator created for a specific transaction is responsible for driving the two-phase commit protocol. In the literature, this is referred to as the *root Transaction Coordinator* or simply root coordinator. Any coordinator that is subsequently created for an existing transaction (for example, as the result of interposition) becomes a subordinate in the process. Such a coordinator is referred to as a *subordinate Transaction Coordinator* or simply subordinate coordinator and by registering a resource becomes a transaction participant. Recoverable servers are always transaction participants. The root coordinator initiates the two-phase commit protocol; participants respond to the operations that implement the protocol. The specification is based on the following rules for commitment and recovery:

1. The protocol defined by this specification is a two-phase commit with presumed rollback.

   This permits efficient implementations to be realized since the root coordinator does not need to log anything before the commit decision and the participants (i.e., *Resource* objects) do not need to log anything before they prepare.

2. *Resource* objects—including subordinate coordinators—do not start commitment by themselves, but wait for prepare to be invoked.

3. The prepare operation is issued at most once to each resource.

4. Participants must remember heuristic decisions until the coordinator or some management application instructs them to forget that decision.

5. A coordinator knows which *Resource* objects are registered in a transaction and so is aware of resources that have completed commitment.

   In general, the coordinator must remember this information if a transaction commits in order to ensure proper completion of the transaction. Resources can be forgotten early if they do not vote to commit the transaction.

6. A participant should be able to request the outcome of a transaction at any time, including after failures occurring subsequent to its *Resource* object being prepared.

7. Participants should be able to report the completion of the transaction (including any heuristic condition).

The recording of information relating to the transaction which is required for recovery is described as if it were a log file for clarity of description; an implementation may use any suitable persistent storage mechanism.

Normal Transaction Completion

Transaction completion can occur in two ways; as part of the normal execution of the Current::commit or Terminator::commit operations or independent of these operations if a failure should occur before normal execution can complete. This section describes the normal (no failure) case. "Failures and Recovery" on page 10-56 describes the failure cases.

Coordinator Role

The root coordinator implements the following protocol:

- When the client asks to commit the transaction, and no prior attempt to rollback the transaction has been made, the coordinator issues the before_completion request to all registered synchronizations.
- When all registered synchronizations have responded, the coordinator issues the prepare request to all registered resources.
- If all registered resources reply VoteReadOnly, then the root coordinator replies to the client that the transaction committed (assuming that the client can still be reached).

  Before doing so, however, it first issues after_completion to any registered synchronizations and, after all responses are received, replies to the client. There is no requirement for the coordinator to log in this case.
- If any registered resource replies VoteRollback or cannot be reached then the coordinator will decide to rollback and will so inform those registered resources which already replied VoteCommit.
- Once a VoteRollback reply is received, a coordinator need not send prepare to the remaining resources. Rollback will be subsequently sent to resources that replied VoteCommit.

  If the report_heuristics parameter was specified on commit, the client will be informed of the rollback outcome when any heuristic reports have been collected (and logged if required).
- Once at least one registered resource has replied VoteCommit and all others have replied VoteCommit or VoteReadOnly, a root coordinator may decide to commit the transaction.
- Before issuing commit operations on those registered resources which replied VoteCommit, the coordinator must ensure that the commit decision and the list of registered resources—those that replied VoteCommit—is stored in stable storage.
- If the coordinator receives VoteCommit or VoteReadOnly responses from each registered resource, it issues the commit request to each registered resource that responded VoteCommit.

- After having received all commit or rollback responses, if synchronizations exist, the root coordinator issues after_completion to each of them passing the transaction outcome as status before responding to the client.
- The root coordinator issues forget to a resource after it receives a heuristic exception.
- This responsibility is not affected by failure of the coordinator. When receiving commit replies containing heuristic information, a coordinator constructs a composite for the transaction.
- The root coordinator forgets the transaction after having logged its heuristic status if heuristics reporting was requested by the originator.
- The root coordinator can now trigger the sending of the reply to the commit operation if heuristic reporting is required. If no heuristic outcomes were recorded, the coordinator can be destroyed.

One Phase Commit

If a coordinator has only a single registered resource, it can perform the commit_one_phase operation on the resource instead of performing prepare and then commit or rollback. If a synchronization exists, before_completion is issued prior to commit_one_phase and after_completion is issued when the response to commit_one_phase has been received. If a failure occurs, the coordinator will not be informed of the transaction outcome.

Subtransactions

When completing a subtransaction, the subtransaction coordinator must notify any registered subtransaction aware resources of the subtransaction's commit or rollback status using the commit_subtransaction or rollback_subtransaction operations of the *SubtransactionAwareResource* interface.

A transaction service implementation determines how it chooses to respond when a resource responds to commit_subtransaction with a system exception. The service may choose to rollback the subtransaction or it may ignore the exceptional condition. The *SubtransactionAwareResource* operations are used to notify the resources of a subtransaction when the subtransaction commits in the case where the resource needs to keep track of the commit status of its ancestors. They are not used to direct the resources to commit or rollback any state. The operations of the *Resource* interface are used to commit or rollback subtransaction resources registered using the register_resource operation of the *Coordinator* interface.

When the subtransaction is committed and after all of the registered subtransaction aware resources have been notified of the commitment, the subtransaction registers any resources registered using register_resource with its parent *Coordinator* or it may register a subordinate coordinator to relay any future requests to the resources.

From the application programmer point of view, the same rules that apply to the completion of top-level transactions also apply to subtransactions. The report_heuristics parameter on commit is ignored since heuristics are not produced when subtransactions are committed.

Recoverable Server Role

A recoverable server includes at least one recoverable object and one *Resource* object. The recoverable object has state that demonstrates at least the atomicity property. The *Resource* object implements the two-phase commit protocol as a participant on behalf of the recoverable object. The responsibilities of each of these objects is described below.

Synchronization Registration

A recoverable server may need to register a *Synchronization* object to ensure that object state data which is persistently managed by a resource is returned to the resource prior to starting the commitment protocol.

Top-Level Registration

A recoverable object registers a *Resource* object with the *Coordinator* so commitment of the transaction including any necessary recovery can be completed.

A recoverable object uses the is_same_transaction operation to determine whether it is already registered in this transaction. It can also use hash_transaction to reduce the number of comparisons. This relies on the definition of the hash_transaction operation to return the same value for all coordinators in the same transaction even if they are generated by multiple Transaction Service implementations.

Once registered, a recoverable server assumes the responsibilities of a transaction participant.

Subtransaction Registration

A Recoverable Server registers for subtransaction completion only if it needs to take specific actions at the time a subtransaction commits. An example would be to change ownership of locks acquired by this subtransaction to its parent.

A recoverable object uses the is_same_transaction operation to determine whether it is already registered in this subtransaction. It can also use hash_transaction to reduce the number of comparisons.

Top Level Synchronization

Synchronization objects ensure that persistent state data is returned to the recoverable object managed by a resource or to the underlying database manager. To do so they implement a protocol which moves the data prior to the prepare phase and does necessary processing after the outcome is complete.

Top-Level Completion

Resource objects implement a recoverable object's involvement in transaction completion. To do so, they must follow the two-phase commit protocol initiated by their coordinator and maintain certain elements of their state in stable storage. The responsibilities of a *Resource* object with regard to a particular transaction depend on how it will vote:

1. Returning VoteCommit to prepare

   Before a *Resource* object replies VoteCommit to a prepare operation, it must implement the following:

   - make persistent the recoverable state of its recoverable object.

     The method by which this is accomplished is implementation dependent. If a recoverable object has only transient state, it need not be made persistent.

   - ensure that its object reference is recorded in stable storage to allow it to participate in recovery in the event of failure.

     How object references are made persistent and then regenerated after a failure is outside the scope of this specification. The Persistent Object Service or some other mechanism may be used. How persistent *Resource* objects get restarted after a failure is also outside the scope of this specification.

   - record the *RecoveryCoordinator* object reference so that it can initiate recovery of the transaction later if necessary.
   - the *Resource* then waits for the coordinator to invoke commit or rollback.
   - A *Resource* with a heuristic outcome must not discard that information until it receives a forget from its coordinator or some administrative component.

2. Returning VoteRollback to prepare

   A *Resource* which replies VoteRollback has no requirement to log. Once having replied, the *Resource* can return recoverable resources to their prior state and forget the transaction.

3. Returning VoteReadOnly to prepare

   A *Resource* which replies VoteReadOnly has no requirement to log. Once having replied, the *Resource* can release its resources and forget the transaction.

Subtransaction Completion

The role of the subtransaction aware resource at subtransaction completion are defined by the subtransaction aware resource itself. The coordinator only requires that it respond to commit_subtransaction or rollback_subtransaction.

All resources need to be notified when a transaction commits or is rolled back. But some resources need to know when subtransactions commit so that they can update local data structures and track the completion status of ancestors. The resource may have rules that are specific to ancestry and must perform some work as all or some ancestors complete. The nested semantics and effort required by the *Resource* object are defined by the object and not the Transaction Service.

Once the resource has been told to prepare, the resource's obligations are exactly the same as a top-level resource.

For example, in the Concurrency Control Service, a resource in a nested transaction might want to know when the subtransaction commits because another subtransaction may be waiting for a lock held by that subtransaction. Once that subtransaction commits, others may be granted the lock. There is no requirement to make lock ownership persistent until a prepare message is received.

For the Persistent Object Service, it is important to keep separate update information associated with a subtransaction. When that subtransaction commits, the Persistent Object Service may need to reorganize its information (such as undo information) in case the parent subtransaction chooses to rollback. Again, the Persistent Object Service resource need not make updates permanent until a prepare message is received. At that point, it has the same responsibilities as a top-level resource.

Subordinate Coordinator Role

An implementation of the Transaction Service may interpose subordinate coordinators to optimize the commit tree for completing the transaction. Such coordinators behave as transaction participants to their superiors and as coordinators to their resources or inferior coordinators.

Synchronization

A subordinate coordinator may register a *Synchronization* object with its superior coordinator if it needs to perform processing before its prepare phase begins.

Registration

A subordinate coordinator registers a *Resource* with its superior coordinator. Once registered, a subordinate coordinator assumes the responsibilities of a transaction participant and implements the behavior of a recoverable server.

Subtransaction Registration

If any of the resources registered with the subordinate coordinator support the *SubtransactionAwareResource* interface, the subordinate coordinator must register a subtransaction aware resource with its parent coordinator. If any of the resources registered with the subordinate using the register_resource operation, the subordinate must register a *Resource* with its superior. If both types of resources were registered with the subordinate, the subordinate only needs to register a subtransaction aware resource with its superior.

Top-level Completion

A subordinate coordinator implements the completion behavior of a recoverable server.

Subtransaction Completion

A subordinate coordinator implements the subtransaction completion behavior of a recoverable server.

Subordinate Coordinator

A subordinate coordinator does not make the commit decision but simply relays the decision of its superior (which may also be a subordinate coordinator) to resources registered with it. A subordinate coordinator acts as a recoverable server as described previously, in terms of saving its state in stable storage. A subordinate coordinator (or indeed any resource) may log the commit decision once it is known (as an optimization) but this is not essential.

- A subordinate coordinator issues the before_completion operation to any synchronizations when it receives prepare from its superior.
- When all responses to before_completion have been received, a subordinate coordinator issues the prepare operation to its registered resources.
- If all registered resources reply VoteReadOnly, then the subordinate coordinator will decide to reply VoteReadOnly.

  Before doing so, however, it first issues after_completion to any registered synchronizations and, after all responses are received, replies VoteReadOnly to its superior. There is no requirement for the subordinate coordinator to log in this case; the subordinate coordinator takes no further part in the transaction and can be destroyed.
- If any registered resource replies VoteRollback or cannot be reached then the subordinate coordinator will decide to rollback and will so inform those registered resources which already replied VoteCommit.

  Once a VoteRollback reply is received, the subordinate coordinator need not send prepare to the remaining resources. The subordinate coordinator issues after_completion to any synchronizations and, after all responses have been received, replies VoteRollback to its superior.
- Once at least one registered resource has replied VoteCommit and all others have replied VoteCommit or VoteReadOnly, a subordinate coordinator may decide to reply VoteCommit.

  The subordinate coordinator must record the prepared state, the reference of its superior *RecoveryCoordinator* and its list of resources that responded VoteCommit in stable storage before responding to prepare.
- A subordinate coordinator issues the commit operation to its registered resources which replied VoteCommit when it receives a commit request from its superior.
- If any resource reports a heuristic outcome, the subordinate coordinator must report a heuristic outcome to its superior.

  Before doing so, however, it first issues after_completion to any registered synchronizations and, after all responses are received, reports the heuristic outcome to its superior. The specific outcome reported depends on the other heuristic outcomes received. The subordinate coordinator must record the heuristic outcome in stable storage.
- After having received all commit replies, a subordinate coordinator logs its heuristic status (if any).

- The subordinate coordinator then replies to the commit from its superior coordinator.

  Before doing so, it issues after_completion to any registered synchronizations and, after all responses have been received, it then replies to its superior. If no heuristic report was sent the *Coordinator* is destroyed.

- A subordinate coordinator performs the rollback operation on its registered resources when it receives a rollback request from its superior.

  If any resource reports a heuristic outcome, the subordinate coordinator records the appropriate heuristic outcome in stable storage and will report this outcome to its superior. Before doing so, however, it issues after_completion to any registered synchronizations and, after receiving all the responses, reports the heuristic outcome to its superior.

- The subordinate coordinator then replies to the rollback from its superior coordinator.

  Before doing so, it issues after_completion to any registered synchronizations and, after all responses have been received, it then replies to its superior. If no heuristic report was sent the *Coordinator* is destroyed.

- If a subordinate coordinator receives a commit_one_phase request, and it has a single registered resource, it can simply perform the commit_one_phase request on its resource. Before doing so, if a synchronization exists, it issues before_completion to the synchronization, then, after receiving the commit_one_phase response, issues after_completion to the synchronization.

  If it has multiple registered resources, it behaves like a superior coordinator, issuing before_completion to any synchronizations and, after receiving the responses, issuing prepare to each resource to determine the outcome, then issuing commit or rollback requests, followed by after_completion requests if synchronizations exist.

- A subordinate coordinator performs the forget operation on those registered resources that reported a heuristic outcome when it receives a forget request from its superior.

Subtransactions

A subordinate coordinator for a subtransaction relays commit_subtransaction and rollback_subtransaction requests to any subtransaction aware resources registered with it. In addition, it performs the same roles as a top-level subordinate coordinator when the top-level transaction commits. It must relay prepare and commit requests to each of the resources that registered with it using the register_resource operation.

Failures and Recovery

The previous descriptions dealt with the protocols associated with the Transaction Service when a transaction completes without failure. To ensure atomicity and durability in the presence of failure, the transaction service defines additional protocols to ensure that transactions, once begun, always complete.

Failure Processing

The unit of failure is termed the failure domain. It may consist of the coordinator and some local resources registered with it, or the coordinator and the resources may each be in its own failure domain.

Local Failure

Any failure in the transaction during the execution of a coordinator prior to the commit decision being made will cause the transaction to be rolled back.

A coordinator is restarted only if it has logged the commit decision.

- If the coordinator only contains heuristic information, nothing is done.
- If the transaction is marked rollback only, a coordinator can send rollback to its resources and inferior coordinators.
- If the transaction outcome is commit, the coordinator sends commit to prepared registered resources and the regular commitment procedure is started.
- If any registered resources exist but cannot be reached, then the coordinator must try again later.

If registered resources no longer exist, then this means that they completed commitment before the coordinator failed and have no heuristic information.

- If a subordinate coordinator is prepared, then it must contact its superior coordinator to determine the transaction outcome.
- If the superior coordinator exists but cannot be reached, then the subordinate must retry recovery later.
- If the superior coordinator no longer exists, then the outcome of the transaction can be presumed to be rollback.

The subordinate will inform its registered resources.

External Failure

Any failure in the transaction during the execution of a coordinator prior to the commit decision being made will cause the transaction to be rolled back.

Transaction Completion after Failure

In general, the approach is to continue the completion protocols at the point where the failure occurred. That means that the coordinator will usually have the responsibility for sending the commit decision to its registered resources. Certain failure conditions will require that the resource initiate the recovery procedure—recall that the resource might also be a subordinate coordinator. These are described in more detail below.

Resources

A resource represents some collection of recoverable data associated with a transaction. It supports the *Resource* interface described in “Resource Interface” on page 10-29. When recovering from failure after its changes have been prepared, a resource uses the replay_completion operation on the *RecoveryCoordinator* to determine the outcome of the transaction and continue completion.

Heuristic Reporting

If the coordinator does not complete the two-phase commit in a timely manner, a subordinate (i.e., a resource or a subordinate coordinator) in the transaction may elect to commit or rollback the resources registered with it in a prepared transaction (take a ***heuristic decision***). When the coordinator eventually sends the outcome, the outcome may differ from that heuristic decision. The result is referred to as HeuristicMixed or HeuristicHazard.The result is reported by the root coordinator to the client only when the report_heuristics option on commit is selected. In these circumstances, the participant (subordinate) and the coordinator must obey a set of rules that define what they report.

Coordinator Role

A root coordinator that fails prior to logging the commit decision can unilaterally rollback the transaction. If its resources have also rolled back because they were not prepared, the transaction is returned to its prior state of consistency. If any resources are prepared, they are required to initiate the recovery process defined below.

- A root coordinator that has a committed outcome will continue the completion protocol by sending commit.
- A root coordinator that has a rolled back outcome will continue the completion protocol by sending rollback.

Synchronizations

Synchronization objects are not persistent so they are not restarted after failure and, as a result, their operations are not invoked during failure processing.

Subtransactions

Subtransactions are not durable, so there is no completion after failure. However, once the top-level coordinator issues prepare, a subtransaction subordinate coordinator has the same responsibilities as a top-level subordinate coordinator.

Recoverable Server role

The Transaction Service imposes certain requirements on the recoverable objects participating in a transaction. These requirements include an obligation to retain certain information at certain times in stable storage (storage not likely to be damaged as the result of failure). When a recoverable object restarts after a failure, it participates in a recovery protocol based on the contents (or lack of contents) of its stable storage.

Once having replied VoteCommit, the resource remains responsible for discovering the outcome of the transaction (i.e., whether to commit or rollback). If the resource subsequently makes a heuristic decision, this does not change its responsibilities to discover the outcome.

If No Heuristic Decision is Made

A resource that is prepared is responsible for initiating recovery. It does so by issuing replay_completion to the *RecoveryCoordinator*. The reply tells the resource the outcome of the transaction. The coordinator can continue the completion protocol allowing the resource to either commit or rollback. The resource can resend replay_completion if the completion protocol is not continued.

- If the resource having replied VoteCommit initiates recovery and receives StExcep::OBJECT_NOT_EXIST, it will know that the *Coordinator* no longer exists and therefore the outcome was to rollback (presumed rollback).
- If the resource having replied VoteCommit initiates recovery and receives StExcep::COMM_FAILURE, it will know only that the *Coordinator* may or may not exist. In this case, the resource retains responsibility for initiating recovery again at a later time.

When a Heuristic Decision is Made

Before acting on a heuristic decision, it must record the decision in stable storage.

- If the heuristic decision turns out to be consistent with the outcome, then all is well and the transaction can be completed and the heuristic decision can be forgotten.
- If the heuristic decision turns out to be wrong, the heuristic damage is recorded in stable storage and one of the heuristic outcome exceptions (HeuristicCommit,HeuristicRollback,HeuristicMixed, or HeuristicHazard) is returned when completion continues.

The heuristic outcome details must be retained persistently until the resource is instructed to forget. In this case, the resource remains persistent until the forget is received.

Subordinate Coordinator Role

The behavior of a subordinate coordinator after a failure of its superior coordinator is implementation-dependent; however, it does follow the following protocols:

- Since it appears as a resource to its superior coordinator, the protocol defined for recoverable servers applies to subordinate coordinators.
- Since it is also a subordinate coordinator for its own registered resources, it is permitted to send duplicate commit, rollback, and forget requests to its registered resources.
- It is required to (eventually) perform either commit or rollback on any resource to which it has received a VoteCommit response to prepare.
- It[1] is required to (eventually) perform the forget operation on any resource that reported a heuristic outcome.

Since subtransactions are not durable, it has no responsibility in this area for failure recovery.

10.5.2 ORB/TS Implementation Considerations

The Transaction Service and the ORB must cooperate to realize certain Transaction Service function. This is discussed in greater detail in the following sections.

Transaction Propagation

The transaction is represented to the application by the *Control* object. Within the Transaction Service, an implicit context is maintained for all threads associated with a transaction. Although there is some common information, the implicit context is not the same as the *Control* object defined in this specification and is distinct from the ORB Context defined by CORBA. It is the implicit context that must be transferred between execution environments to support transaction propagation.

The objects using a particular Transaction Service implementation in a system form a Transaction Service domain. Within the domain, the structure and meaning of the implicit context information can be private to the implementation. When leaving the domain, this information must be translated to a common form if it is to be understood by the target Transaction Service domain, even across a single ORB. When the implicit context is transferred, it is represented as a *PropagationContext*.

No OMG IDL declaration is required to cause propagation of the implicit context with a request. The minimum amount of information that could serve as an implicit context is the object reference of the *Coordinator*. However, an identifier (e.g., an X/Open XID) is also required to allow efficient (local) execution of the is_same_transaction and hash_transaction operations when interposition is done. Implementations may choose to also include the *Terminator* object reference if they support the ability for ending the transaction in other execution environments than the originator's. Transferring the implicit context requires interaction between the Transaction Service and the ORB to add or extract the implicit context from ORB messages. This interaction is also used to implement the checking functions described in "X/Open Checked Transactions" on page 10-37.

When the *Control* object is passed as an operation argument (explicit propagation), no special transfer mechanism is required.

Interposition

When a transaction is propagated, the implicit context is exported and can be used by the importing Transaction Service implementation to create a new *Control* object which refers to a new (local) *Coordinator*. This technique, *interposition*, allows a

1. or some "agent" acting on its behalf: for example a system management application.

surrogate to handle the functions of a coordinator in the importing domain. These coordinators act as subordinate coordinators. When interposition is performed, a single transaction is represented by multiple *Coordinator* objects.

Interposition allows cooperating Transaction Services to share the responsibility for completing a transaction and can be used to minimize the number of network messages sent during the completion process. Interposition is required for a Transaction Service implementation to implement the is_same_transaction and hash_transaction operations as local method invocations, thus improving overall systems performance.

An interposed coordinator registers as a participant in the transaction with the *Coordinator* identified in the *PropagationContext* of the received request. The relationships between coordinators in the transaction form a tree. The root coordinator is responsible for completing the transaction.

Many implementations of the Transaction Service will want to perform interposition and thus create *Control* objects and subsequently *Coordinator* objects for each execution environment participating in the transaction. To create a new (local) *Control*, an importing Transaction Service uses the information in the propagation context to recreate a *Control* object using a *TransactionFactory*. Interposition must be complete before the get_control operation can complete in the target object. An object adaptor is one possible place to implement interposition.

Subordinate Coordinator Synchronization

A subordinate coordinator may register with its superior coordinator to ensure that any local state data maintained by the subordinate coordinator is returned to the underlying resource prior to the subordinate coordinator's associated *Resource* seeing prepare.

Subordinate Coordinator Registration

A subordinate coordinator must register with its superior coordinator to orchestrate transaction completion for its local resources. The register_resource operation of the *Coordinator* can be used to perform this function. The subordinate coordinator can either support the *Resource* interface itself or provide another *Resource* object which will support transaction completion. Some implementations of the Transaction Service may wish to perform this function as a by-product of invoking the first operation on an object in a new domain as is done with the X/Open model. This requires that the information necessary to perform registration be added to the reply message of that first operation.

Transaction Service Interoperation

The Transaction Service can be implemented by multiple components at different locations. The different components can be based on the same or different implementations of the Transaction Service. As stated in "Principles of Function, Design, and Performance" on page 10-8, it is a requirement that multiple Transaction Services interoperate across the same ORB and different ORBs.

Transaction Service interoperation is specified by defining the data structures exported between different implementations of the Transaction Service. When the implicit context is propagated with a request, the destination uses it to locate the superior coordinator. That coordinator may be implemented by a foreign Transaction Service. By registering a resource with that coordinator, the destination arranges to receive two-phase commit requests from the (possibly foreign) Transaction Service.

The Transaction Service permits many configurations; no particular configuration is mandated. Typically, each program will be directly associated with a single Transaction Service. However, when requests are transmitted between programs in different Transaction Service domains, both Transaction Services must understand the shared data structures to interoperate.

An interface between the ORB and the Transaction Service is defined that arranges for the implicit context to be carried on messages that represent method invocations made within the scope of a transaction.

Structure of the Propagation Context

The *PropagationContext* structure is defined in “Structures” on page 10-15. For the functions defined within the base section of the propagation context, it is necessary only to send it with requests. Implementations may use the vendor specific portion for additional functions (for example, to register an interposed coordinator with its superior), which may require the propagation context to be returned. Whether it is returned or not, is implementation specific.

otid_t

The otid_t structure is a more efficient OMG IDL version of the X/Open defined transaction identifier (XID). The otid_t can be transformed to an X/Open XID and vice versa.

TransIdentity

A structure that defines information for a single transaction. It consists of a coord, an optional term, and an otid.

coord

The *Coordinator* for this transaction in the exporting Transaction Service domain.

term

The *Terminator* for this transaction in the exporting Transaction Service domain. Transaction Services that do not allow termination by other than the originator will set this field to a null reference (OBJECT_NIL).

otid

An identifier specific to the current transaction or subtransaction. This value is intended to support efficient (local) execution of the is_same_transaction and hash_transaction operations when the importing Transaction Service does interposition.

timeout

The timeout value associated with the transaction in the relevant set_timeout operation (or the default timeout).

<TransIdentity> parents

A sequence of TransIdentity structures representing the parent(s) of the current transaction. The ordering of the sequence starts at the parent of the current transaction and includes all ancestors up to the top-level transaction. An implementation that does not support nested transactions would send an empty sequence. This allows a non-nested transaction implementation to know when a nested transaction is being imported. It also supports efficient (local) execution of the *Coordinator* operations which test parentage when the importing Transaction Service does interposition.

implementation_specific_data

This information is exported from an implementation and is required to be passed back with the rest of the context if the transaction is re-imported into that implementation.

Appearance of the Propagation Context in Messages

The appearance of the *PropagationContext* in messages is defined by the CORBA interoperability specification (see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*). The Transaction Service passes the PropagationContext to the ORB via the *TSPortability* interface defined in "The Transaction Service Callbacks" on page 10-65.

- When exporting a transaction, the ORB sets the PropagationContext into the ServiceContext::context_data field and marshals the PropagationContext as defined by the GIOP message format and marshalling rules.
- When importing a transaction, the ORB demarshalls the ServiceContext::context_data according to the GIOP formatting rules and extracts the PropagationContext to be presented to the Transaction Service.

For more information, see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*.

Transaction Service Portability

This section describes the way in which the ORB and the Transaction Service cooperate to enable the *PropagationContext* to be passed and any X/Open-style checking to be performed on transactional requests.

Because it is recognized that other object services and future extensions to the CORBA specification may require similar mechanisms, this component is specified separately from the main body of the Transaction Service to allow it to be revised or replaced by a mechanism common to several services independently of any future Transaction Service revisions.

To enable a single Transaction Service to work with multiple ORBs, it is necessary to define a specific interface between the ORB and the Transaction Service, which conforming ORB implementations will provide, and demanding Transaction Service implementations can rely on. The remainder of this section describes these interfaces. There are two elements of the required interfaces:

1. An additional ORB interface that allows the Transaction Service to identify itself to the ORB when present in order to be involved in the transmission of transactional requests.
2. A collection of Transaction Service operations (the Transaction Service callbacks) that the ORB invokes when a transactional request is sent and received.

These interfaces are defined as pseudo-IDL to allow them to be implemented as procedure calls.

Identification of the Transaction Service to the ORB

Prior to the first transactional request, the Transaction Service will identify itself to the ORB within its domain to establish the transaction callbacks to be used for transactional requests and replies.

The Transaction Service identifies itself to the ORB using the following interface.

```
interface TSIdentification { // PIDL
    exception NotAvailable {};
    exception AlreadyIdentified {};

    void identify_sender(in CosTSPortability::Sender sender)
        raises (NotAvailable, AlreadyIdentified);
    void identify_receiver(in CosTSPortability::Receiver receiver)
        raises (NotAvailable, AlreadyIdentified);
};
```

The callback routines identified in this operation are always in the same addressing domain as the ORB. On most machine architectures, there are a unique set of callbacks per address space. Since invocation is via a procedure call, independent failures cannot occur.

NotAvailable

The NotAvailable exception is raised if the ORB implementation does not support the *CosTSPortability* module.

AlreadyIdentified

The AlreadyIdentified exception is raised if the identify_sender or identify_receiver operation had previously identified callbacks to the ORB for this addressing domain.

identify_sender

The identify_sender operation provides the interface that defines the callbacks to be invoked by the ORB when a transactional request is sent and its reply received.

identify_receiver

The identify_receiver operation provides the interface that defines the callbacks to be invoked by the ORB when a transactional request is received and its reply sent.

The Transaction Service must identify itself to the ORB at least once per Transaction Service domain. Sending and receiving transactional requests are separately identified. If the callback interfaces are different for different processes within a Transaction Service domain, they are identified to the ORB on a per process basis. Only one Transaction Service implementation per addressing domain can identify itself to the ORB.

A Transaction Service implementation that only sends transactional request can identify only the sender callbacks. A Transaction Service that only receives transactional requests can identify only the receiver callbacks.

The Transaction Service Callbacks

The *CosTSPortability* module defines two interfaces. Both interfaces are defined as PIDL. The *Sender* interface defines a pair of operations which are called by the ORB sending the request before it is sent and after its reply is received. The *Receiver* interface defines a pair of operations which are called by the ORB receiving the request when the request is received and before its reply is sent. Both interfaces use the *PropagationContext* structure defined in "Structures" on page 10-15.

```
module CosTSPortability { // PIDL
	typedef long ReqId;

	interface Sender {
		void sending_request(in ReqId id,
			out CosTransactions::PropagationContext ctx);
		void received_reply(in ReqId id,
			in CosTransactions::PropagationContext ctx,
			in CORBA::Environment env);
	};

	interface Receiver {
		void received_request(in ReqId id,
			in CosTransactions::PropagationContext ctx);
		void sending_reply(in ReqId id,
			out CosTransactions::PropagationContext ctx);
	};
};
```

ReqId

The ReqId is an unique identifier generated by the ORB which lasts for the duration of the processing of the request and its associated reply to allow the Transaction Service to correlate callback requests and replies.

Sender::sending_request

A request is about to be sent. The Transaction Service returns a *PropagationContext* to be delivered to the Transaction Service at the server managing the target object. The TRANSACTION_REQUIRED standard exception is raised when invoked outside the scope of a transaction.

Sender::received_reply

A reply has been received. The *PropagationContext* from the server is passed to the Transaction Service along with the returned environment. The Transaction Service examines the Environment to determine whether the request was successfully performed. If the Environment indicates the request was unsuccessful, the TRANSACTION_ROLLEDBACK standard exception is raised.

Receiver::received_request

A request has been received. The *PropagationContext* defines the transaction making the request. It is associated with the target object only if the target object inherits from the *TransactionalObject* interface.

Receiver::sending_reply

A reply is about to be sent. A checking transaction service determines whether there are outstanding deferred requests or subtransactions and raises a system exception using the normal mechanisms. The exception data from the callback operation needs to be re-raised by the calling ORB.

Behavior of the Callback Interfaces

The following sections describe the protocols associated with the callback interfaces:

Requirements on the ORB

The ORB will invoke the sender callbacks only when a transactional operation is issued for an object in a different process. Objects within the same process implicitly share the same transaction context. The receiver callbacks are invoked when the ORB receives a transactional request from a different process.

The ORB must generate a request identifier for each outgoing request and be able to associate the identifier with the reply when it is returned. For deferred synchronous invocations, this allows the Transaction Service to correlate the reply with the request to implement checked behavior. The request identifier is passed on synchronous invocations to permit the same interface to be used.

The callbacks are invoked in line with the processing of requests and replies. This means that the callbacks will be executed on the same thread that issued or processed the actual request or reply. When the DII is used, the received_reply callback must be invoked on the same thread that will subsequently process the response.

Requirements on the Transaction Service

Within a single process, the transaction context is part of the thread specific state. Multiple threads executing on behalf of the same transaction will share the same transaction context since a thread can only execute on behalf of a single transaction at a time. Since the callbacks are defined as PIDL (procedure calls), they are invoked on the client's thread when sending and the server's thread when receiving. This enables the Transaction Service to locate the proper transaction context when sending and associate the received transaction context with the thread that will process the transactional operation. The callback interfaces may only raise standard exceptions and may not make additional object invocations using the ORB.

10.5.3 Model Interoperability

The indirect context management programming model of the Transaction Service is designed to be compatible with the X/Open DTP standard, and implementable by existing Transaction Managers. In X/Open DTP, a current transaction is associated with a ***thread of control***. Some X/Open Transaction Managers support a single thread of control in a ***process***, others allow multiple threads of control per process.

Model interoperability is possible because the Transaction Service design is compatible with the X/Open DTP model of a Transaction Manager. X/Open associates an implicit current transaction with each thread of control.

This means that a single transaction management service can provide the interfaces defined for the Transaction Service and also provide the TX and XA interfaces of X/Open DTP. This is illustrated in Figure 10-6.

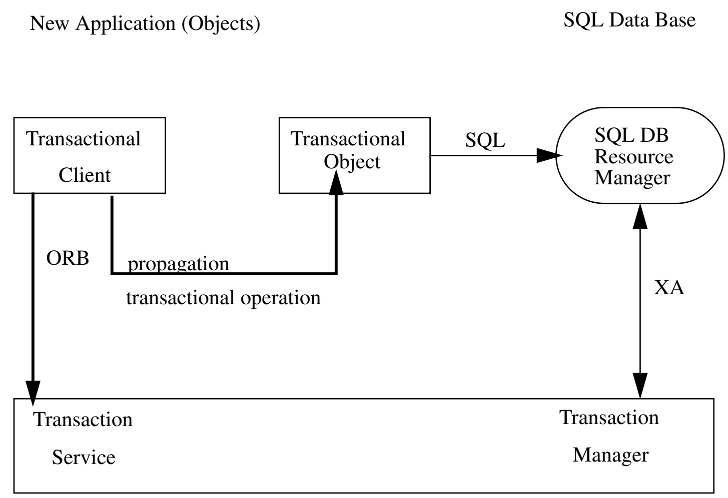


Figure 10-6 Model Interoperability Example

The transactional object making the SQL call, and the SQL Resource manager, are both executing on the same thread of control. The transaction manager is able to recognize the relationship between the transaction context of the object, and the transaction associated with the SQL DB.

The *Current* and *Coordinator* interfaces of the Transaction Service implement two-phase commit for the objects in the transaction. The Resource Manager will participate in the two-phase commitment process via the X/Open XA interface.

10.6 The CosTransactions Module

```
#include <Corba.idl>
module CosTransactions {
// DATATYPES
enum Status {
	StatusActive,
	StatusMarkedRollback,
	StatusPrepared,
	StatusCommitted,
	StatusRolledBack,
	StatusUnknown,
	StatusNoTransaction,
	StatusPreparing,
	StatusCommitting,
	StatusRollingBack
};

enum Vote {
	VoteCommit,
	VoteRollback,
	VoteReadOnly
};

// Structure definitions
struct otid_t {
	long formatID; /*format identifier. 0 is OSI TP */
	long bqual_length;
	sequence <octet> tid;
};
struct TransIdentity {
	Coordinator coord;
	Terminator term;
	otid_t otid;
};
struct PropagationContext {
	unsigned long timeout;
	TransIdentity current;
	sequence <TransIdentity> parents;
	any implementation_specific_data;
};

// Forward references for interfaces defined later in module
interface Current;
interface TransactionFactory;
interface Control;
interface Terminator;
interface Coordinator;
```

```
interface RecoveryCoordinator;
interface Resource;
interface Synchronization;
interface SubtransactionAwareResource;
interface TransactionalObject;

// Heuristic exceptions
exception HeuristicRollback {};
exception HeuristicCommit {};
exception HeuristicMixed {};
exception HeuristicHazard {};

// Other transaction-specific exceptions
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};

// Current transaction
interface Current : CORBA::Current {
	void begin()
		raises(SubtransactionsUnavailable);
	void commit(in boolean report_heuristics)
		raises(
			NoTransaction,
			HeuristicMixed,
			HeuristicHazard
		);
	void rollback()
		raises(NoTransaction);
	void rollback_only()
		raises(NoTransaction);

	Status get_status();
	string get_transaction_name();
	void set_timeout(in unsigned long seconds);
	Control get_control();
	Control suspend();
	void resume(in Control which)
		raises(InvalidControl);
};
```

```
interface TransactionFactory {
        Control create(in unsigned long time_out);
        Control recreate(in PropagationContext ctx);
};

interface Control {
        Terminator get_terminator()
                raises(Unavailable);
        Coordinator get_coordinator()
                raises(Unavailable);
};

interface Terminator {
        void commit(in boolean report_heuristics)
                raises(
                        HeuristicMixed,
                        HeuristicHazard
                );
        void rollback();
};

interface Coordinator {

        Status get_status();
        Status get_parent_status();
        Status get_top_level_status();

        boolean is_same_transaction(in Coordinator tc);
        boolean is_related_transaction(in Coordinator tc);
        boolean is_ancestor_transaction(in Coordinator tc);
        boolean is_descendant_transaction(in Coordinator tc);
        boolean is_top_level_transaction();

        unsigned long hash_transaction();
        unsigned long hash_top_level_tran();

        RecoveryCoordinator register_resource(in Resource r)
                raises(Inactive);

        void register_synchronization (in Synchronization sync)
                raises(Inactive, SynchronizationUnavailable);

        void register_subtran_aware(in SubtransactionAwareResource r)
                raises(Inactive, NotSubtransaction);

        void rollback_only()
                raises(Inactive);

        string get_transaction_name();
        Control create_subtransaction()
                raises(SubtransactionsUnavailable, Inactive);
```

```
        PropagationContext get_txcontext ()
                raises(Unavailable);
};

interface RecoveryCoordinator {
        Status replay_completion(in Resource r)
                raises(NotPrepared);
};

interface Resource {
        Vote prepare()
                        raises(
                        HeuristicMixed,
                        HeuristicHazard
                );
        void rollback()
                raises(
                        HeuristicCommit,
                        HeuristicMixed,
                        HeuristicHazard
                );
        void commit()
                raises(
                        NotPrepared,
                        HeuristicRollback,
                        HeuristicMixed,
                        HeuristicHazard
                );
        void commit_one_phase()
                raises(
                        HeuristicHazard
                );
        void forget();
};

interface TransactionalObject {
};

interface Synchronization : TransactionalObject {
        void before_completion();
        void after_completion(in Status status);
};

interface SubtransactionAwareResource : Resource {
        void commit_subtransaction(in Coordinator parent);
        void rollback_subtransaction();
};

}; // End of CosTransactions Module
```

10.6.1 The CosTSPortability Module

```
module CosTSPortability { // PIDL
    typedef long ReqId;

    interface Sender {
        void sending_request(in ReqId id,
            out CosTransactions::PropagationContext ctx);
        void received_reply(in ReqId id,
            in CosTransactions::PropagationContext ctx,
            in CORBA::Environment env);
    };

    interface Receiver {
        void received_request(in ReqId id,
            in CosTransactions::PropagationContext ctx);
        void sending_reply(in ReqId id,
            out CosTransactions::PropagationContext ctx);
    };
};
```

Appendix A Relationship of Transaction Service to TP Standards

This appendix discusses the relationship and possible interactions with the following related standards:

- X/Open TX interface
- X/Open XA interface
- OSI TP protocol
- LU 6.2 protocol
- ODMG standard

A.1 Support of X/Open TX Interface

A.1.1 Requirements

The X/Open DTP model[1] is now widely known and implemented.

Since the Transaction Service and the X/Open DTP models are interoperable, an application using transactional objects could use the TX interface, the X/Open-defined interface to delineate transactions, to interact with a Transaction Manager. (The Transaction Manager is the access point of the Transaction Service.)

A.1.2 TX Mappings

The correspondence between the TX interface primitives and the Transaction Service operations (*Current* interface) are as follows:

Table 10-2 TX mappings

TX interface	*Current* interface
tx_open()	*no equivalent*
tx_close()	*no equivalent*
tx_begin()	Current::begin()
tx_rollback()	Current::rollback() or Current::rollback_only()
tx_commit()	Current::commit()
tx_set_commit_return()	report_heuristics parameter of Current::commit()
tx_set_transaction_control()	*no equivalent (chained transactions not supported)*
tx_set_transaction_timeout()	Current::set_timeout()

1. See "Distributed Transaction Processing: The XA Specification, X/Open Document C193." X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.

Table 10-2 TX mappings

TX interface	*Current* interface
tx_info() - XID	Coordinator::get_txcontext() Current::get_name()[1]
tx_info() - COMMIT_RETURN	*no equivalent*
tx_info() - TRANSACTION_TIME_OUT	*no equivalent*
tx_info() - TRANSACTION_STATE	Current::get_status()

1. A printable string is output: not guaranteed to be the XID in all implementations.

tx_open

tx_open() provides a way to open, in a given execution environment, the Transaction Manager and the set of Resource Managers that are linked to it. Such an operation does not exist in the Transaction Service; such processing may be implicitly executed when the first operation of the Transaction Service is executed in the execution environment.

This processing is also related to a future Initialization Service.

tx_close

tx_close() provides a way to close, in a given execution environment, the Transaction Manager and the set of Resource Managers that are linked to it. Such an operation does not exist in the Transaction Service.

tx_begin

tx_begin() corresponds to Current::begin() or to TransactionFactory::create().

tx_rollback

tx_rollback() corresponds to Current::rollback(), Terminator::rollback(), Current::rollback_only() or Coordinator::rollback_only() . In TX, when a server calls tx_rollback(), the transaction may be rolled back or set as rollback only, as in the Transaction Service.

tx_commit and tx_set_commit_return

tx_commit() corresponds to Current::commit(. The Transaction Service operations have a parameter, report_heuristics, corresponding to the commit_return parameter of TX.

tx_set_transaction_control

tx_set_transaction_control() is used, in TX, to switch between unchained and chained mode; this function is not needed in the Transaction Service environment because it does not support chained transactions.

tx_set_transaction_timeout

tx_set_transaction_timeout() corresponds to Current::set_timeout() or TransactionFactory::create().

tx_info

tx_info() returns information related to the current transaction. In the Transaction Service:

- the XID may be retrieved by Coordinator::get_txcontext();
- the XID (in effect) may be retrieved by Current::get_transaction_name();
- the transaction state may be retrieved by Current::get_status();
- the commit return attribute is not needed because this attribute is given in the commit() operation;
- the timeout attribute cannot be obtained.

A.2 Support of X/Open Resource Managers

A.2.1 Requirements

X/Open DTP-compliant Resource Managers, simply called X/Open Resource Managers or RMs, are Resource Managers that can be involved in a distributed transaction by allowing their two-phase commit protocol to be controlled via the X/Open XA Interface. Many RDBMS suppliers currently offer (or intend to offer) X/Open Resource Managers. Many OODBMS' intend also to support the XA Interface (some have already implemented it).

The Transaction Service must therefore be able to interact with X/Open Resource Managers. This section will illustrate how an X/Open Resource Manager may be used by a Transaction Service-compliant system.

The architecture of Transaction Service, based on the same concepts as the X/Open DTP Model, allows mapping of Transaction Service operations to and from XA interactions.

A.2.2 XA Mappings

This section gives an overall view of a possible mapping between XA primitives offered by an X/Open Resource Manager (called RM hereafter) and the interfaces of the Transaction Service and their operations in the different phases of a transaction and during recovery.

The mappings are summarized in the following table:

Table 10-3 XA mappings

X/Open	Object Transaction Service
xa_start() ax_reg()	Receiver::received_request Current::resume
xa_end()	Receiver::sending_reply Current::suspend
ax_unreg()	*no equivalent*
xa_prepare()	Resource::prepare
xa_commit()	Resource::commit
xa_rollback()	Resource::rollback
xa_recover()	*no equivalent*
no equivalent	RecoveryCoordinator::replay_completion()
xa_forget()	Resource::forget()

In the X/Open DTP model all the interactions are made in the same X/Open thread of control.

A.2.3 XID

An XID is the Transaction Identifier. As defined by X/Open, this XID is the only information used by Resource Managers to associate logged information to the transaction, including objects' before images, after images, locks, and transaction state.

The contents of an XID is defined by X/Open as follows:

```
#define XIDDATASIZE 128 /* size in bytes */
#define MAXGTRIDSIZE 64
      /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE 64
      /* maximum size in bytes of bqual */

struct xid_t {
     long formatID;/* format identifier */
     long gtrid_length;
            /* value not to exceed 64 */
     long bqual_length;
            /* value not to exceed 64 */
     char data [XIDDATASIZE];
};
typedef struct xid_t XID;
```

The XID uniquely and unambiguously identifies a distributed transaction (information contained in the gtrid part of the XID) and a transaction-branch, the work performed by a node in the transaction tree (information contained in the bqual part of the XID).

To facilitate the use of distributed transaction in heterogeneous environments, X/Open has adopted the structure of the Transaction Identifier used in OSI TP but allows the use of other Transaction Identifiers formats, which may be defined by the value of a Format Identifier field contained in the XID structure. The OSI TP Transaction Identifier contains information about the initiator of the transaction and the superior in the transaction tree; this information may be used, during recovery, to contact these entities and obtain the outcome of the transaction.

In the Transaction Service, tightly-coupled concurrency is assumed (a lock held by a transaction may be accessed by any participant of the same transaction) and the transaction branch part of the XID must not be given to RMs.

Interactions with an XA-compliant RM

Model

To model the relationship between the XA interface and the Transaction Service operation, an X/Open Transaction Manager has been modeled; this component is used here as a way to describe the interactions and may be implemented in a different manner.

Propagation of a Transaction to an RM

An RM may support two kinds of involvement interactions:

- Static registration, in which the Transaction Service involves the RM whenever it is itself involved in a new transaction.
- Dynamic registration, in which the RM notifies the Transaction Service that it has been requested to perform some work and request the XID of the current transaction.

An RM gets involved in a transaction when it has to perform some new work for this transaction. This happens in one of the following situations:

- A request carrying a transaction context has just been received and the RM has to perform work for the target object of this request;
- A method performing a request that is carrying a transaction context is resumed (by a Current::resume() operation).

An object may receive several requests carrying a transaction context for the same transaction. An RM may also perform work for several objects in the same transaction. Thus an RM may be involved several times in the same transaction; the “resume” and the “join” concepts of XA may be used to notify the RM of any multiple involvement. When an RM has to get involved in a transaction, it must obtain the corresponding XID from the Transaction Service through an xa_start() primitive or by a return parameter of an ax_reg() primitive. This XID is transmitted to the RM as a parameter to xa_start() or ax_reg() and is used by the RM to relate any work performed or any lock obtained to the transaction.

If the Transaction Service is called by an ax_reg() while it is not aware of any transaction, it returns a null XID to the RM. The RM is then free to start a local transaction of its own, and no Transaction Service transaction will be accepted until the RM issues an ax_unreg().

Refer to X/Open documents for more information about propagation of a transaction to an RM.

First phase of Commitment

When the first phase of commitment is started, the Transaction Service issues an xa_prepare() primitive and process its results to determine its decision.

Second Phase of Commitment

When the second phase of commitment is started, the Transaction Service issues an xa_commit() primitive and process its results to determine the heuristic situation.

One-phase commitment

When the Transaction Service wants to perform a one-phase commitment, it issues an xa_commit() primitive and process its results to determine the heuristic situation.

In the XA interface, there is no specific primitive for one-phase commitment: an RM must consider an xa_commit() without preceding xa_prepare() as a request to perform a one-phase commitment.

Rollback

When a rollback has to be performed, the Transaction Service issues an xa_rollback() primitive and process its results to determine the heuristic situation.

Recovery

In the XA interface, the recovery of an RM is triggered by the Transaction Manager which issues an xa_recover(); the RM then gives back a list of all XIDs that are either in the Ready state or have been heuristically completed.

In the Transaction Service recovery is performed by a resource that issues a replay_completion operation to a *Coordinator* (see Subsection "Transaction Completion after Failure" in “Transaction Service Protocols” on page 10-49).

Failure of an Operation

Any failure of an operation typically leads to a rollback of the transaction, especially if it is not possible to determine whether the operation has been performed or not. However, in the decided commit state, the commit operation must be retried until the reply has been received (unless a heuristic hazard condition is detected).

Failure of an RM

If an RM fails, the Transaction Service detecting the failure will issue an xa_recover(). The Transaction Service will then get a list of XIDs of transactions for which the RM is in the ready state and transactions that have been heuristically completed.

The Transaction Service will then:

- Call xa_rollback() for all transactions that it knows to be neither in the prepared state nor in the decided commit state.
- Call xa_commit() for all transactions that it knows to be in the decided commit state.
- Wait for the decisions commit or rollback for the other.

Failure of Transaction Service

Upon warm restart of the Transaction Service and retrieval of the states of transactions needing recovery from stable storage, the Transaction Service will call xa_recover() to get the list of transactions for which the RM needs recovery (see failure of an RM, here above).

A.3 Interoperation with Transactional Protocols

A.3.1 Transactional Protocols

A CORBA application may sometimes need to interoperate with one or more applications using one of the de-facto standard transactional protocol: OSI TP and SNA LU 6.2. In this case, the Transaction Service must be able to import or export transactions using one of these protocols.

Export is the ability to relate a transaction of the Transaction Service to a transaction of a foreign transactional protocol. Importing means relating a Transaction Service transaction to a transaction started on a remote application and propagated via the foreign transactional protocol.

Since the model used by the Transaction Service is similar to the model of OSI TP and the X/Open DTP model, the interactions with OSI TP are straightforward. Since OSI TP is a compatible superset of SNA LU 6.2, a mapping to SNA communications is easily accomplished.

To interoperate, a mapping should be defined for the two-phase commit, rollback, and recovery mechanisms, and for the transaction identifiers.

Notice that neither OSI TP nor SNA LU 6.2 supports nested transactions.

A.3.2 OSI TP Interoperability

OSI TP [ISO92] is the transactional protocol defined by ISO. It has been selected by X/Open to allow the distribution of transactions by one of the communication interfaces: remote procedure call[1], client-server [2] or peer-to-peer (CPI-C Level-2 API [CIW93]).

The Transaction Service supports only unchained transactions. The use of dialogues using the Chained Transactions functional unit is possible only if restrictive rules are defined. These rules are not described in this document.

OSI TP Transaction Identifiers

In OSI TP, loosely-coupled transactions are supported and every node of the transaction tree possesses a transaction branch identifier which is composed of the transaction identifier (or atomic action identifier) and a branch identifier (the branch identifier being null for the root node of the transaction tree). Both the transaction identifier and the branch identifier contains an AE-Title (Application Entity Title) and a suffix that make it unique within a certain scope.

The format of the standard X/Open XID is compatible with the OSI TP identifiers, the gtrid corresponding to the atomic action identifier and the bqual corresponding to the branch identifier.

Incoming OSI TP Communications (Imported Transactions)

The Transaction Service is a subordinate in an OSI TP transaction tree and interacts with its superior by regular PDUs as defined by the OSI TP protocol. The Transaction Service introduces the transaction identifier received on the OSI TP dialogue using the TransactionFactory::recreate operation.

The Transaction Service maps the OSI TP commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an OSI TP Prepare message, will enter the first phase of commitment procedure.
- When it enters the prepared state for the transaction, the Transaction Service will trigger the sending of an OSI TP Ready message to its superior. (It may trigger a Recover (Ready) message when normal communications are broken with the superior).
- The Transaction Service, upon reception of an OSI TP Commit message, enters the second phase of commitment procedure. (It may receive a Recover (Commit) when normal communications are broken with the superior.)
- The Transaction Service, upon reception of an OSI TP Rollback message (it may be a Recover (Unknown) when normal communications are broken with the superior or any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last rollback reply, will trigger the sending of a Rollback Response/Confirm message to its superior.

1. See “Distributed Transaction Processing: The TxRPC Specification, X/Open Document P305.” X/Open Company Ltd., Reading, U.K..

2. See “Distributed Transaction Processing: The XATMI Specification, X/Open Document P306.” X/Open Company Ltd., Reading, U.K..

Outgoing OSI TP Communications (Exported Transactions)

The Transaction Service behaves as a superior in an OSI TP transaction tree and interacts with its subordinates by regular PDUs as defined by the OSI TP protocol.

The Transaction Service will map the OSI TP commitment procedure as follows:

- The Transaction Service, during the first phase of commitment procedure will invoke an OSI TP Prepare message to all its subordinates.
- Upon reception of an OSI TP Ready message, the Transaction Service will process this message as a successful reply to prepare.
- The Transaction Service, upon entering the second phase of the commitment procedure will send an OSI TP Commit message (it may be a Recover (Commit) when normal communications are broken with the subordinate) to all subordinates.
- The Transaction Service, upon reception of an OSI TP Rollback message (it may be any other rollback-initiating condition) will enter its rollback procedure (unless a rollback is already in progress).
- The Transaction Service, upon reception of the last Rollback Response/Confirm message from its subordinates, will process this message as a reply to a rollback operation and determine the heuristic situation.

A.3.3 SNA LU 6.2 Interoperability

SNA LU 6.2 ([SNA88a], [SNA88b]) is a transactional protocol defined by IBM. It is widely used for transaction distribution. The standard interface to access LU 6.2 communications is CPI-C (Common Programming Interface for Communications) defined by IBM in the context of SAA [CPIC93] and currently being evolved by the CPI-C Implementers' Workshop to become CPI-C level 2, a modern interface usable for LU 6.2 and OSI TP communications [CIW93].

LU 6.2 supports only chained transactions but, at a given node, a transaction is started only when resources have been involved in the transaction. LU 6.2 can be used for a portion of an “unchained” transaction tree if the LU 6.2 conversations are ended after each transaction by any node that has both LU 6.2 conversations and dialogues of an unchained transaction.

LU 6.2 Transaction Identifiers

SNA LU 6.2 also supports loosely-coupled transactions and uses a specific format for transaction identifiers: the Logical Unit of Work (LUWID) corresponds to the OSI Transaction Identifier. The LUWID is composed of:

- The Fully Qualified Logical Unit Name, which is composed of up to 17 bytes, is unique in an SNA network or a set of interconnected SNA networks.
- An instance number which is unique at the LU that create the transaction.
- The sequence number that is incremented whenever the transaction is committed.

The Conversation Correlator corresponds to the OSI TP Branch Identifier; it is a string of 1 to 8 bytes which are unique within the context of the LU having established the conversation and is meaningful when combined with the Fully Qualified LU Name of this Logical Unit.

Incoming LU 6.2 Communications

The LU 6.2 two-phase commit protocol is different from the OSI TP protocol: the system sending a Prepare message has to perform logging and is responsible for recovery. LU 6.2 does also support features like last-agent optimization, read-only and allows any node in the transaction tree to request commitment.

The Transaction Service is a subordinate in an LU 6.2 transaction tree and interacts with its superior using SNA requests and responses as defined by the LU 6.2 protocol. The Transaction Service maps the LUWID corresponding to the incoming conversation to an OMG otid_t and issues TransactionFactory::recreate to import the transaction.

The Transaction Service maps the LU 6.2 commitment, rollback and recovery procedures to the Transaction Service commitment procedure as follows:

- The Transaction Service, upon reception of an LU 6.2 Prepare message will enter the first phase of commitment procedure.
- The Transaction Service, upon entering the prepared state for the transaction, the Transaction Service will trigger the sending of a Request Commit message to is superior.
- The Transaction Service, upon reception of an LU 6.2 Committed message (it may be a Compare States (Committed) when normal communications are broken with the superior) will enter the second phase of commitment procedure.
- The Transaction Service, upon leaving the decided commit state, will trigger the sending of a Forget message to is superior (it may be a Reset when normal communications are broken with the superior).

Due to the two-phase commit difference, the Transaction Service will never send the equivalent of the Recover(Ready) unless prompted by the superior.

The last-agent and read-only features may also be supported by the Transaction Service.

Outgoing LU 6.2 Communications

The Transaction Service has to log when the Prepare message is sent and, in case of communication failure or restart of the Transaction Service, a recovery is needed.

ODMG Standard

ODMG-93 is a standard defined by ODMG (Object Database Management Group) describing portable interface to access Object Database Management Systems (ODBMS).

Since it is likely that, in the future, many objects involved in transactions will be handled by an ODBMS, this standard has a strong relationship with the Transaction Service.

A.4 ODMG Model

The ODMG model defines optional transactions and supports the nested transaction concept. The ODMG model does not cover the integration of ODBMS with an external Transaction Service, allowing other resources and communications to be involved in a transaction. No two-phase commit or recovery protocol is described.

A transaction object must be created. The transactional operations are:

- Begin (or start) to begin a transaction (or a subtransaction).
- Commit to request commitment of a transaction.
- Abort to rollback a transaction.
- Checkpoint to commit the transaction but keep the locks. This feature is not supported by the current version of the Transaction Service.
- abort_to_top_level to request rollback of a nested transaction family. The Transaction Service does not directly support this feature but does provide means to perform this functionality by resuming the context of the top-level transaction and then requesting rollback.

If the transaction object is destroyed, the transaction is rolled back.

Integration of ODMG ODBMSs with the Transaction Service

Since ODMG-93 does not define any way to integrate an ODBMS into an existing transaction, the integration is difficult unless the ODBMS supports the XA interface, in which case the section on XA-compliant RM is applicable.

In the future, it is anticipated that ODBMS will implement the Transaction Service-defined interfaces and be considered as a recoverable server.

A possibility is to use, at a root node, an ODBMS as a last resource and, after all subordinates are prepared, to request a one-phase commitment to the ODBMS. If the outcome for the ODBMS is commit, the transaction will be committed, if it is rollback, the transaction will be rolled back. The mechanism may work if it is possible to determine, after a crash, whether the ODBMS committed or rolled back; this may be done at application level.

Appendix B Transaction Service Glossary

B.1 TransactionTerms

2PC: See *Two-phase commit.*

Abort: See *Rollback*

Active: The state of a transaction when processing is in progress and *completion* of the transaction has not yet commenced.

Atomicity: A transaction property that ensures that if work is interrupted by failure, any partially completed results will be undone. A transaction whose work completes is said to commit. A transaction whose work is completely undone is said to rollback (abort).

Begin: An operation on the Transaction Service which establishes the initial boundary of a transaction.

Commit: Commit has two definitions as follows:

An operation in the *Current* and *Terminator* interfaces that a program uses to request that the current transaction terminate normally and that the effects of that transaction be made permanent.

An operation in the *Resource* interface which causes the effects a transaction to be made permanent.

Commit coordinator: In a two-phase commit protocol, the program that collects the vote from the participants.

Commit participant: In a two-phase commit protocol, the program that returns a vote on the completion of a transaction.

Committed: The property of a transaction or a transactional object, when it has successfully performed the commit protocol. See also *in-doubt*, *active*, and *rolled back*.

Completion: The processing required (either by *commit* or *rollback*) to obtain the durable outcome of a transaction.

Coordinator: A coordinator involves *Resource* objects in a transaction when they are registered. A coordinator is responsible for driving the two-phase commit protocol. See also *Commit coordinator* and *Commit participant*.

Consistency: A property of a transaction that ensures that the transaction's actions, taken as a group, do not violate any of the integrity constraints associated with the state of its associated objects. This requires that the application program be implemented correctly: the Transaction Service provides the functionality to support application data consistency.

Decided commit state: A root coordinator enters the decided commit state when it has written a log-commit record; a subordinate coordinator or resource is in the decided commit state when it has received the commit instruction from its superior; in the latter case, a log-commit record may be written but this is not essential.

Decided rollback state: A coordinator or resource enters the decided rollback state when it decides to rollback the transaction or has received a signal to do so.

Direct context management: An application manipulates the *Control* object and the other objects associated with the transaction. See also *Indirect context management.*

Durability: A transaction property that ensures the results of a successfully completed transaction will never be lost, except in the event of catastrophe. It is generally implemented by a combination of persistent storage and a logging service that provides a backup copy of permanent changes.

Execution environment: An implementation-dependent factor that may determine the outcome of certain operations on the Transaction Service. Typically the execution environment is the scope within which shared state is managed.

Flat Transaction: A transaction that has no subtransactions—and that cannot have subtransactions.

Forgotten "state": This is not really a transaction state at all, because there is no memory of the transaction: it has either completed or rolled back and all records on permanent storage have been deleted.

Heuristic Commit or Rollback: To unilaterally make the commit or rollback decision about *in-doubt* transactions when the coordinator fails or contact with the coordinator fails.

Indirect context management: An application uses the *Current* object, provided by the Transaction Service, to associate the transaction context with the application thread of control. See also *Direct context management.*

In-doubt: The state of a transaction if it is controlled by a transaction manager that can not be contacted, so the commit decision is in doubt. See also *active*, *committed*, *rolled back*.

Interposition: Adding a sequence of one or more *subordinate coordinators* between a *root coordinator* and its participants.

Isolation: A transaction property that allows concurrent execution, but the results will be the same as if execution was serialized. Isolation ensures that concurrently executing transactions cannot observe inconsistencies in shared data.

Lock service: Called the Concurrency Control Service, it is an Object Service used by resources to control access to shared objects by concurrently executing methods.

Log-ready record (and contents): for an intermediate coordinator a log-ready record contains identification of the (superior) coordinator and of *Resource* objects (including subordinate coordinators) registered with the coordinator which replied VoteCommit (i.e., it excludes registered objects which replied VoteReadOnly); for a *Resource* object a log-ready record includes identification of the coordinator with which it is registered.

Log-commit record (and contents): A log-commit record contains identification of all registered *Resource* objects which replied VoteCommit.

Log-heuristic record: This contains a record of a heuristic decision either HeuristicCommit or HeuristicRollback.

Log-damage record: This contains a record of heuristic damage i.e. where it is known that a heuristic decision conflicted with the decided outcome (HeuristicMixed) or where there is a risk that a heuristic decision conflicted with the decided outcome (HeuristicHazard).

Log service: A service used by resource managers for recording recovery information and the Transaction Service for recording transaction state durably.

Nested transaction: A transaction that either has subtransaction or is a subtransaction on some other transaction.

Participant: See *Commit participant.*

Persistent storage: Generally speaking, a synonym for *Stable storage*. In the context of the OMA, the Persistent Object Service (POS) provides an object representation of stable storage.

Prepared: The state that a transaction is in when phase one of a two-phase commit has completed.

Presumed rollback: An optimization of the two-phase commit protocol that results in more efficient performance as the *root coordinator* does not need to log anything before the commit decision and the *Participants* (i.e. *Resource* objects) do not need to log anything before they prepare. So called because, at restart, if no record of the transaction is found, it is safe to assume the transaction rolled back.

Propagation: A function of the Transaction Service that allows the *Transaction context* of a client to be associated with a transactional operation on a server object. The Transaction Service supports both implicit and explicit propagation of transaction context.

Recoverable Object: An object whose data is affected by committing or rolling back a transaction.

Recoverable Server: A transactional object with recoverable state that registers a *Resource* (not necessarily itself) with a *Coordinator* to participate in transaction completion.

Recovery Service: A service used by resource managers for restoring the state of objects to a prior state of consistency.

Resource: An object in the Transaction Service that is registered for involvement in two-phase commit—*2PC*. Corresponds to a *Resource Manager.*

Resource Manager: An X/Open term for a component which manages the integrity of the state of a set of related resources.

Rollback: Rollback (also known as *Abort*) has two definitions, as follows:

An operation in the *Current* and *Terminator* interfaces used to indicate that the current transaction has terminated abnormally and its effects should be discarded.

An operation in the *Resource* interface which causes all state changes in the transaction to be undone.

Rolled Back: The property of a transaction or a transactional object when it has discarded all changes made in the current transaction. See also *in-doubt*, *active*, and *committed*.

Root Coordinator: The first coordinator in a sequence of coordinators where there is interposition. The coordinator associated with the transaction originator.

Security Service: An object service which provides identifications of users (authentication), controls access to resources (authorization), and provides auditing of resource access.

Stable storage: Storage not likely to be damaged as the result of node failure.

Sub-coordinator: See *Subordinate Coordinator.*

Subordinate Coordinator: A coordinator subordinate to the *root coordinator* when *interposition* has been performed. A subordinate coordinator appears as a *Resource* object to its superior. Also known as a *Sub-coordinator.*

Synchronization: An object in the Transaction Service which controls the transfer of persistent object state data so it can be made durable by its associated resource.

Thread: The entity that is currently in control of the processor.

Thread Service: A service which enables methods to be executed concurrently by the same process. Where two or more methods can execute concurrently each method is associated with its own thread of control.

TP monitor: A system component that accepts input work requests and associates resources with the programs that act upon these requests to provide a run-time environment for program execution.

Transaction: A collection of operations on the physical and abstract application state.

Transactional client: An arbitrary program that can invoke operations of many transactional objects in a single transaction. Not necessarily the *Transaction originator.*

Transaction Context: The transaction information associated with a specific thread. See *Propagation*.

Transactional operation: An operation on an object that participates in the propagation of the current transaction.

Transaction originator: An arbitrary program—typically, a transactional client, but not necessarily an object—that begins a transaction.

Transaction Manager: A system component that implements the protocol engine for 2-phase commit protocol. See also *Transaction Service*.

Transactional object: An object whose operations are affected by being invoked within the scope of a transaction.

Transactional server: A collection of one or more objects whose behavior is affected by the transaction, but has no recoverable state of its own.

Transaction Service: An Object Service that implements the protocols required to guarantee the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions. See also *Transaction Manager.*

TSPortability: An interface of the Transaction Service which allows it to track transactional operations and propagate transaction context to another Transaction Service implementation.

Two-Phase commit: A transaction manager protocol for ensuring that all changes to recoverable resources occur atomically and furthermore, the failure of any resource to complete will cause all other resource to undo changes. Also called *2PC*.

Query Service Specification 11

11.1 Service Description

11.1.1 Overview

The Query Service provides query operations on collections of objects. The queries are predicate-based and may return collections of objects. They can be specified using object derivatives of SQL and/or other styles of object query languages, including direct manipulation query languages.

The term “query” has read-only connotations, but we use it to denote general manipulation operations including selection, insertion, updating and deletion on collections of objects. Throughout this chapter, the term “object” is used in the general sense to include data.

The Query Service can be used to return collections of objects that may be:

- Selected from source collections based on whether their member objects satisfy a given predicate.
- Produced by query evaluators based on the evaluation of a given predicate. These query evaluators may manage implicit collections of objects.

The source and result collections may be typed. The source collection may be specified by the client or may be the result of previous queries.

11.1.2 Design Principles

The Query Service exists to allow arbitrary users and objects to invoke queries on arbitrary collections of other objects. Such queries are declarative statements with predicates, including the ability to specify values of attributes; to invoke arbitrary operations; and to invoke arbitrary services within the OMG environment, such as the Life Cycle, Persistent Object, and Relationship Services.

To support the OMG architecture, the Query Service must allow querying against any objects, with arbitrary attributes and operations.

To be useful in practical situations, the Query Service must allow use of performance enhancing mechanisms, such as indexing.

To be useful in environments with database systems—object-oriented, relational, and other—and with other systems that store and access large collections of objects, the Query Service must map well to these native systems' internal mechanisms for specifying collections and using indexing. The Query Service must also allow the native systems to contribute to specifying collections and indexing.

To maximize usefulness to the community at large, the Query Service is based on existing standards for query and extended when necessary to accommodate other design principles.

The Query Service also supports flexibility in implementation and extensions.

11.1.3 Architecture

The Query Service design provides an architecture for a nested and federated service that can coordinate multiple nested query evaluators, much as the Transaction Service provides an architecture for a nested and federated service that can coordinate multiple nested resources managers.

Query Evaluators: Nesting and Federation

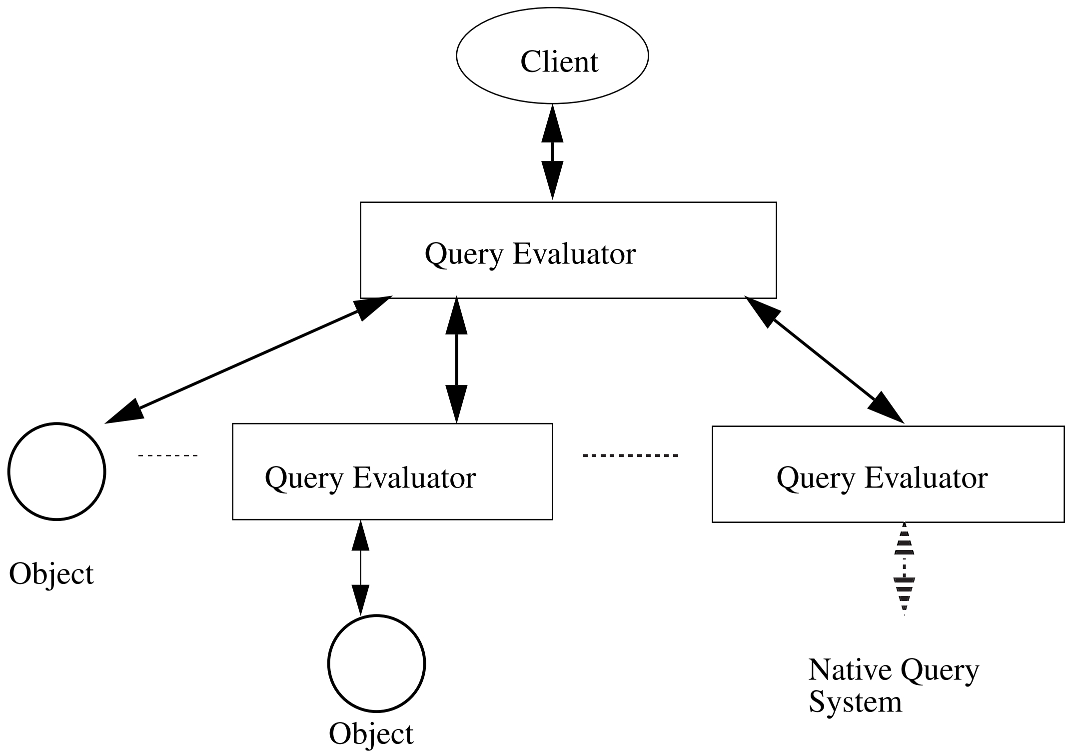


Figure 11-1 Query Evaluators: Nesting and Federation

Objects may participate in the Query Service in two ways. The simplest involves any CORBA object as is. The Query Evaluator is then responsible for evaluating the query predicate and performing all query operations by invoking operations on that object through its published OMG IDL interfaces. Any non-supported operations trigger exceptions. This mechanism provides the greatest generality, including support for all CORBA objects, but with the least optimization.

In a more involved manner, objects participate as members of a collection, either explicit or implicit. The collection supports a specific query interface (that is, the collection is itself a Query Evaluator). In this case, the Query Evaluator passes the query predicate to the collection, which then evaluates the predicate and performs query operations on an appropriate member object, receives any result, combines such results with all other participating object results, and returns this to the caller. This accomplishes the nesting, by passing the query evaluation on to a lower level. Such nesting may continue to an arbitrary number of levels, without limit.

This second way allows Query Evaluators or any associated native query systems to evaluate the query using the internal optimization at their disposal. This is expected to include faster access, caching, and indexing. Interpretation of names embedded in query predicates is determined by the Query Evaluator or its associated native query systems.

The Query Service specification does not define evaluation, indexing or optimization mechanisms. These are in the province of the implementor and may vary significantly in different environments. The Query Service simply provides a mechanism for passing the query to such systems and allowing their optimizations to take effect.

Collections

The Query Service provides definitions and interfaces for creating and manipulating collections of objects. These (explicit) collections may form both the scope to which a query may be applied and the result of the query, when the result is one or more objects.

The collections are defined as objects, with methods for adding and removing members. They may be arbitrary in nature. In particular, they are not limited to type extents, as in some object systems, though type extents are examples of such collections. They may map directly to collections managed by native query systems, for optimization, and may also include arbitrary CORBA objects.

Associated iterators are defined to allow manipulation of collections, including traversal over and retrieval of the objects within the collections. Such iterators allow a constant interface that can be invoked and implemented for arbitrary situations, including mixtures of general CORBA objects; native query system collections; highly distributed collections that could not be simultaneously accessed; collections across multiple heterogeneous products and systems; very small collections; and very large collections that could not be materialized physically.

Queryable Collections for Scope and Result

For collections to serve as both the result of a query and as a scope for another query, these collections must themselves be Query Evaluators. Such collections are called Queryable Collections. They support both the Query Evaluator and collection interfaces, as illustrated in Figure 11-2.

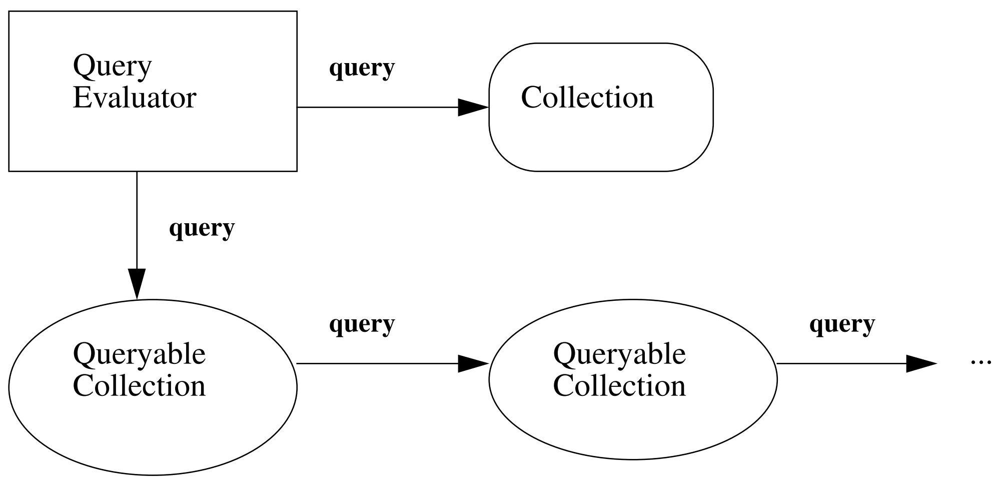


Figure 11-2 Queryable Collections

One of the issues that arises in using Queryable Collections is scoping in a nested environment. If the collection being queried allows adding arbitrary objects, and if objects are then added which are outside the scope of the evaluation mechanism of the Queryable Collection, then the Queryable Collection would have to provide the full functionality of a top-level Query Evaluator, evaluating predicates on arbitrary CORBA objects. This would defeat the purpose of nesting.

To solve this problem, we allow Queryable Collection implementations, in response to the invocation of the add and replace operations, to internally decide whether to add or replace the specified object, and to raise an exception if they decide not to. This allows arbitrary Queryable Collections—which are always supported at the top Query Evaluator level, and sublevel implementations that scope Queryable Collections to their own domain—to use whatever local mechanisms their (possibly pre-existing) query engines use. Examples of local mechanisms include optimization capabilities such as physical and logical indices; clustering; caching, and so forth.

Query Objects

Since queries can be complex and resource-demanding, there are numerous circumstances under which one would like to:

- Use graphical means to construct a query.

- Save a query and re-execute it later on, maybe with different set of search parameters.
- Precompile a query for later execution; this may be for the purpose of syntax and semantics checking and/or query optimization.
- Execute a query in an asynchronous manner; go do something else and come back for the result.
- Check the status of a long-running query and decide whether to continue or abort.

The Query Service provides the preceding capabilities and extensions through the use of Query objects. A Query object is created by calling a Query Manager, which is a more powerful form of Query Evaluator. Once created, a client of the Query object can:

- Use whatever means appropriate to construct the query specification.
- Prepare the query for later execution.
- Execute the query any number of times, with the same or different set of search parameters.
- Check the status of the query.
- Obtain the result of the query.

How the Query object does the preceding tasks is determined by the Query object and its associated Query Manager.

11.1.4 Query Languages

By using a very general model and by using predicates to deal with queries, the Query Service is designed to be independent of any specific query languages. Therefore, a particular Query Service implementation can be based on a variety of query languages and their associated query processors.

However, in order to provide query interoperability among the widest variety of query systems and to provide object-level query interoperability, a Query Service provider must support one of the following two query languages: SQL Query or OQL.

(Query capability is commonly implemented in database systems, hence there are many products, tools, trained users, and experiences based on these implementations. To leverage this, we base the query language specification on SQL Query and OQL.)

- **SQL Query.** Specifically, SQL-92 Query, which is defined in Chapter 7 (Entry SQL), and Sections 13.7, 13.8 and 13.10 (Entry SQL) of Reference 1 on page 11-27. SQL Query is used as the generic term to denote the evolution of SQL-92 Query. That is, it is envisioned that SQL-92 Query will evolve into SQL-9x Query, and so forth. These will be future versions of SQL Query. SQL-92 Query is the current version.

- **OQL.** Specifically, OQL-93, which is defined in Chapter 4 of Reference 4 on page 11-27. OQL is used as the generic term to denote the evolution of OQL-93. That is, it is envisioned that OQL-93 will evolve into OQL-9x, and so on. These will be future versions of OQL. OQL-93 is the current version.

For those Query Service providers who intend to provide only basic object-level query interoperability (for example, to support the needs of the Life Cycle Service or Property Service), the following must also be supported:

- **OQL Basic**. Specifically, OQL-93 Basic, which is defined in Sections 4.11.1.2, 4.11.1.3, 4.11.1.4, 4.11.1.5, 4.11.1.6 (**set** only), 4.11.1.7 (except **first** and **last**) and 4.11.1.10 in Reference 4 on page 11-27.

Ideally we would like to specify a single query language, for complete query interoperability. The most widely used query language in currently available query systems is SQL-92 Query, which does not support full object query capabilities. OQL-93 does support full object query capabilities and contains a near- (but not exact) subset of SQL-92 Query. Including SQL-92 Query provides the widest interoperability with the most query systems, while including OQL-93 provides full OMG Object Model support and full object query capabilities.

X3H2 and ODMG have started working together toward merging SQL Query and OQL with the goal of specifying a single standard query language. As SQL Query and OQL evolve, the OMG will revise of the Query Service to conform to future changes.

SQL Query

In the relational database world the accepted standard for database language is SQL-92 (Reference 1 on page 11-27). The ANSI X3H2 committee is working on a new version, SQL3 (Reference 5 on page 11-27), which will include object extensions, among other things. The committee is still working on the details of the modeling constructs; the object model under consideration is different from the OMG's Object Model. It is important for the eventual SQL object model to be fully compatible with the OMG Object Model so that SQL Query, the query subset of SQL, can serve as the query *lingua franca* in the OMG environment.

SQL-92 is a full database language. Functionally, it consists of the following types of language statements: schema; data; transaction; commection; session; dynamic; diagnostics; and embedded exception declaration. Among these, only a subset of data statements deal directly with query. This subset is defined to be SQL-92 Query. SQL-92 Query basically deals with query over tables (special kind of collections) of rows (special kind of dynamic data structures). As such, it concerns with a sub-domain of object query.

OQL

In the object database world the leading standard is ODMG-93 (Reference 4 on page 11-27). The ODMG-93 standard includes an object model, based on the OMG's Core Object Model, with extensions, to form the proposed object database profile. Also included is the Object Definition Language, ODL, which is a strict superset of IDL, providing a means to define objects in this profile model. All extensions, including attributes and relationships, are visible in the object interfaces as operations, and hence remain compatible with OMG IDL and the OMG architecture.

ODMG-93 also includes OQL (that is, OQL-93). OQL-93 is an adaptation of the SQL-92 Query capability to extend to all objects in the ODMG object model. It includes the ability to include operation invocation in queries, to query over object inheritance hierarchies, to invoke inter-object relationships, and to query over arbitrary collections. OQL-93 is a query-only language; that is, it allows evaluation of a predicate and a returned result, but includes no specific constructs for object modification. The ability within OQL-93 to invoke operations provides the insert, update and delete capability without violating encapsulation.

The OQL-93 syntax and semantics are not exactly compatible with SQL-92 Query. However, ODMG is working with X2H2 to address this issue. It is important for the eventual OQL to be fully compatible with SQL Query so that there is only one standard query language. .

SQL Query = OQL

Both X3H2 and ODMG have agreed upon a vision of the evolution of SQL Query and OQL, as illustrated in Figure 11-3.

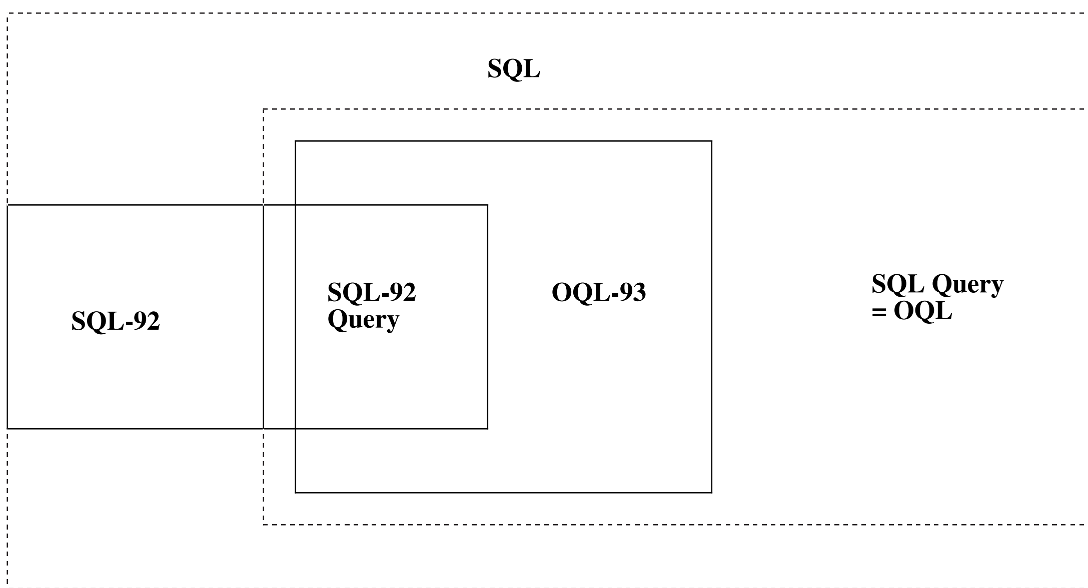


Figure 11-3 SQL Query = OQL

In Figure 11-3, solid lines indicate existing, defined specifications, while dotted lines indicate future specifications. As can be seen, SQL-92 Query is the query portion of SQL-92. OQL-93, being a query only language and having object features, overlaps with SQL-92 and is almost exactly compatible with it.

SQL-92 will evolve toward a future SQL, which is a full database language. OQL-93 will evolve toward a future OQL. The agreement from X3H2 and ODMG is to make the query subset of SQL, SQL Query, and OQL identical so that there is a single, common query language specification.

11.1.5 Key Features

The following are key features of the Query Service:

- Provides operations of selection, insertion, updating, and deletion on collections of objects. The objects may be transient or persistent, local or remote; the objects may have arbitrary attributes and operations.
- Accommodates different granularity of objects accessed by queries, including good support for high performance access to fine-grained objects.
- Allows the scope of the objects accessible in and via the collections that are the immediate operands of the query operations.
- Supports querying and/or returning complex data structures.
- Supports operating on user defined collections of objects.
- Supports operating on other kinds of collections and sets.
- Allows the use of attributes, inheritance, and procedurally-specified operations in the query predicate and in the computation of results.
- Allows the use of available interfaces defined by OMG-adopted specifications.
- Allows the use of relationships for navigation, including testing for the existence of a relationship between objects.
- Does not require breaking the encapsulation provided by the interfaces to objects.

In addition, the Query Service:

- Provides an extensible framework for dealing with object query.
- Is independent of the specific syntax and semantics of the query language used. The query language can be SQL Query, OQL, a graphical query language, or any other suitable object query language. In order to provide query interoperability among the widest variety of query systems and object-level query interoperability, a Query Service provider must support either SQL Query or OQL (OQL Basic with basic object-level interoperability) as specified in Section 11.1.4 on page 11-6.
- Allows for associative query and navigational query.

11.2 Service Structure

11.2.1 Overview

The Query Service defines two types of service. The specification is organized around these types.

Type One: Collections

The *Collection* and *Iterator* interfaces define the interfaces to create and manipulate collections of objects. The *Collection* interface is defined with operations for adding, retrieving, replacing, and removing member objects. The collections that it represents may be arbitrary in nature. The *Iterator* interface is defined with operations for traversing over and retrieving objects within a collection.

Type Two: Query Framework

The Query Framework interfaces define a flexible and extensible framework for dealing with object query. The *QueryLanguageType* interface provides the scheme to use the OMG IDL type system to classify query language types. The *QueryEvaluator* interface defines the basic operation to evaluate a query. The result of the query, which can serve as the scope for further queries, is represented by the *QueryableCollection*. The *QueryManager* interface defines a more powerful QueryEvaluator which can be called upon to create arbitrary *Query* objects. Such objects can provide the capability for graphical query construction, pre-compilation and optimization, asynchronous query execution, and so forth.

11.2.2 Collection Interface Structure

The collection interfaces are arranged into the interface structure illustrated in Figure 11-4. Dotted arrows represent association.

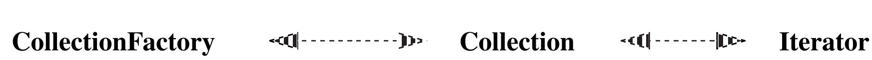


Figure 11-4 Collection interface structure

11.2.3 Query Framework Interface Hierarchy/Structure

The query framework interfaces are arranged into the interface hierarchy/structure illustrated in Figure 11-5. Solid arrows represent inheritance and dotted arrows represent association.

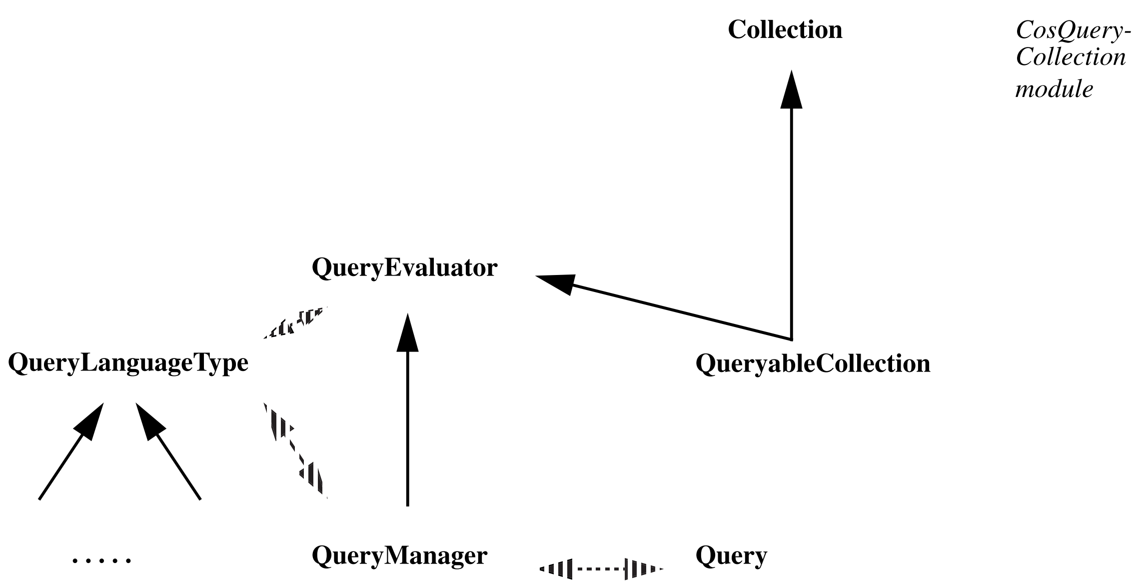


Figure 11-5 Query Framework interface hierarchy/structure

11.2.4 Interface Overview

The Query Service defines the interfaces to support the functionality described in Section 11.1 on page 11-1.

Table 11-1 and Table 11-2 give high level summaries of the Query Service interfaces. Collection interfaces are described in detail starting in the section Section 11.3 on page 11-12. Query interfaces are described in Section 11.5 on page 11-19.

Table 11-1 Interfaces defined in the *CosQueryCollection* module

Interface	**Purpose**
CollectionFactory	To create collections
Collection	To aggregate objects
Iterator	To iterate over collections

Table 11-2 Interfaces defined in the *CosQuery* module

Interface	Purpose
QueryLanguageType and its subtypes	To represent query language types
QueryEvaluator	To evaluate query predicates and execute query operations
QueryableCollection	To represent the scope and result of queries
QueryManager	To create query objects and perform query processing
Query	To represent queries

11.3 The Collection Model

11.3.1 Common Types of Collections

The Collection interface allows you to manipulate objects in a group. The objects that are part of a Collection are called its *elements*. Examples of common types of Collections are as follows:

- An *Equality Collection* has elements that can be checked for equality among each other. An example is a set.
- A *Key Collection* uses keys to identify elements (a key is part of an element). An example is a key bag.
- An *Ordered Collection* has its elements arranged so that there is always a first element, last element, next element, and previous element. Ordered Collections can be further classified as one of the following types:
  - A *Sequential Collection* has sequentially ordered elements. An example is a sequence.
  - A *Sorted Collection* has sorted elements. An example is a sorted set (which is also an equality Collection).

The Query Service defines only a top-level, basic Collection interface that supports query on arbitrary collections without restriction to any particular type. Subtyping can be used to map this basic Collection interface into a variety of collection classes, including the ANSI C++ Standard Template Library (STL), ODMGs, and others. The OMG Collection Service, available in the future, is expected to fit in similarly well.

11.3.2 Iterators

An Iterator is a movable pointer into a Collection. An Iterator is created in association with a Collection and can be used by a client to move through the member elements of the Collection. When an Iterator is created for an ordered Collection, it points to the beginning or the first element of the Collection. A series of next operations move it through subsequent elements until it passes through the last element and points to the end of the Collection. For unordered Collections, the elements are visited in an arbitrary order. Each element is visited exactly once.

The Iterator interface allows traversing a Collection in a way that works consistently for arbitrarily large Collections. In addition to the next operation, which can be used to move through the next element, it provides a reset operation to restart the iteration. Multiple Iterators can be created to maintain state concerning traversal of the same or different Collections.

The behavior of an Iterator can become undefined if elements are added to or deleted from its associated Collection. This means that its behavior depends upon the type and implementation of the Collection. In particular, an Iterator may become invalid as a result of such actions. Once an Iterator becomes invalid, it must be reset before it can be used for traversal again.

11.4 The CosQueryCollection Module

The *CosQueryCollection* module defines the Collection interfaces of the Query Service. In particular, it defines the

- *CollectionFactory* interfaces, to create Collections.
- *Collection* interface, to represent generic collections.
- *Iterator* interface, to enumerate the Collections.

The *CosQueryCollection* module is shown below.

```
module CosQueryCollection {

        exception ElementInvalid {};
        exception IteratorInvalid {};
        exception PositionInvalid {};

        enum ValueType {TypeBoolean, TypeChar, TypeOctet, TypeShort, TypeUShort,
TypeLong, TypeULong, TypeFloat, TypeDouble, TypeString, TypeObject, TypeAny,
TypeSmallInt, TypeInteger, TypeReal, TypeDoublePrecision, TypeCharacter, TypeDecimal,
TypeNumeric};
        struct Decimal {long precision; long scale; sequence<octet> value;}
        union Value switch(ValueType) {
                case TypeBoolean: boolean b;
                case TypeChar:   char c;
                case TypeOctet: octet o;
                case TypeShort : short s;
                case TypeUShort : unsigned short us;
                case TypeLong : long l;
                case TypeULong : unsigned long ul;
                case TypeFloat : float f;
                case TypeDouble : double d;
                case TypeString : string str;
                case TypeObject : Object obj;
                case TypeAny : any a;
                case TypeSmallInt : short si;
                case TypeInteger : long i;
                case TypeReal : float r;
                case TypeDoublePrecision : double dp;
                case TypeCharacter : string ch;
                case TypeDecimal : Decimal dec;
                case TypeNumeric : Decimal n;
        };
        typedef boolean Null;
        union FieldValue switch(Null) {
                case false : Value v;
        };
        typedef sequence<FieldValue> Record;

        typedef string Istring;
        struct NVPair {Istring name; any value;};
        typedef sequence<NVPair> ParameterList;

        interface Collection;
```

Figure 11-6 CosQueryCollection Module

```
module CosQueryCollection {
        interface Iterator;

        interface CollectionFactory {
                Collection create (in ParameterList params);
        };

        interface Collection {
                readonly attribute long cardinality;

                void add_element (in any element) raises(ElementInvalid);
                void add_all_elements (in Collection elements) raises(ElementInvalid);

                void insert_element_at (in any element, in Iterator where) raises(IteratorInvalid,
ElementInvalid);

                void replace_element_at (in any element, in Iterator where) raises(IteratorInvalid,
PositionInvalid, ElementInvalid);

                void remove_element_at (in Iterator where) raises(IteratorInvalid, PositionInvalid);
                void remove_all_elements ();

                any retrieve_element_at (in Iterator where) raises(IteratorInvalid, PositionInvalid);

                Iterator create_iterator ();
        };


        interface Iterator {
                any next () raises(IteratorInvalid, PositionInvalid);

                void reset ();
                boolean more ();
        };
};
```

Figure 11-6 CosQueryCollection Module

11.4.1 The CollectionFactory Interface

The *CollectionFactory* interface defines an operation for creating an instance of a Collection.

Creating a Collection

```
Collection create (in ParameterList params);
```

This operation creates a new instance of a Collection. The factory is passed a list of parameters, one of which must be:

"initial_size", type long

which represents an initial, estimated number of elements. The Collection is initially empty and may grow dynamically, both in elements and size. Other parameters that may be passed include, for example, "hints" relating to indexing, and so forth.

The ParameterList is defined to be a sequence of name-value pairs, of which the name is defined to be of type Istring. As is the case in the Naming Service, Istring is a placeholder for a future OMG IDL internationalized string data type.

11.4.2 The Collection Interface

The *Collection* interface defines operations to:

- Add elements
- Replace elements
- Remove elements
- Retrieve elements

to and from a collection and an operation to create iterators for traversing the collection.

The element type of a collection can be any. This is designed to accommodate generality. For most common queries, the result collections tend to consist of elements that are records or objects. For some specific queries, however, the result collections may consist of elements of any data type.

Record is defined to be a sequence of FieldValues. A FieldValue may be Null or may have a value. This is designed to provide direct mapping to similar features available in a wide variety of existing query systems. The type of a FieldValue can be one of the OMG IDL base types, string, Object or one of the suggested mappings to SQL data types: TypeSmallInt; TypeInteger; TypeReal; TypeDoublePrecision; TypeCharacter; TypeDecimal; and TypeNumeric. (TypeFloat is the same as that defined for the OMG IDL base type.)

Determining the Cardinality

```
readonly attribute long cardinality;
```

This attribute identifies the number of elements that a Collection contains.

Adding an Element

```
void add_element (in any element) raises(ElementInvalid);
```

This operation adds an element to a Collection. Behaviors of all Iterators of the Collection become undefined when the element is added.

A Collection implementation, in response to the invocation of the add_element() operation, may internally decide whether to add the specified element, raising the ElementInvalid exception if it decides not to add it. As discussed in "Queryable Collections for Scope and Result" on page 11-5, this allows sublevel Query Evaluator implementations that scope Queryable Collections to their own domain.

Adding Elements from a Collection

```
void add_all_elements (in Collection elements) raises
(ElementInvalid);
```

This operation adds all elements of the input Collection to a Collection. The elements are added in the Iterator order of the input Collection and are consistent with the semantics of add_element(). This operation is really a sequence of add_element(). If any elements are added, behaviors of all Iterators of the Collection become undefined.

Inserting an Element

```
void insert_element_at (in any element, in Iterator where)
raises(IteratorInvalid, ElementInvalid);
```

This operation inserts an element to a Collection at the position pointed to by the input Iterator. Behaviors of all Iterators of the Collection, except the input Iterator, become undefined when the element is inserted.

If the input Iterator is invalid, the IteratorInvalid exception will be raised. The ElementInvalid exception will be raised as it is for the add_element() operation.

Replacing an Element

```
void replace_element_at (in any element, in Iterator where)
raises(IteratorInvalid, PositionInvalid, ElementInvalid);
```

This operation replaces the element of a Collection, pointed to by the input Iterator, with the input element. The input element must have the same positioning property as the replaced element. (Only equality Collections and key Collections have positioning property.)

If the input Iterator is invalid, the IteratorInvalid exception will be raised. If the Iterator does not point at an element, the PositionInvalid exception will be raised. The ElementInvalid exception will be raised in the same manner as it is for the add_element() operation.

Removing an Element

```
void remove_element_at (in Iterator where) raises
(IteratorInvalid, PositionInvalid);
```

This operation removes the element of a Collection, pointed to by the input Iterator. After removal, behaviors of all Iterators of the Collection become undefined.

If the input Iterator is invalid, the IteratorInvalid exception will be raised. If the Iterator does not point at an element, the PositionInvalid exception will be raised.

Removing all Elements

```
void remove_all_elements ();
```

This operation removes all elements from a Collection. After removal, behaviors of all Iterators of the Collection become undefined.

Retrieving an Element

```
any retrieve_element_at (in Iterator where) raises
(IteratorInvalid, PositionInvalid);
```

This operation retrieves the element of a Collection, pointed to by the input Iterator.

If the input Iterator is invalid, the IteratorInvalid exception will be raised. If the Iterator does not point at an element, the PositionInvalid exception will be raised.

Creating an Iterator

```
Iterator create_iterator ();
```

This operation creates an Iterator for a Collection. The Iterator is initially set at the beginning of the Collection.

11.4.3 The Iterator Interface

The Iterator interface defines operations to:

- Access and navigate through elements of a collection
- Reset the iteration
- Test for completion of an iteration

Accessing the Current Element

```
any next () raises(IteratorInvalid, PositionInvalid);
```

This operation retrieves the element of a Collection, pointed to by the Iterator, and advances the Iterator position. The operation will raise the IteratorInvalid exception if the Iterator is invalid, and the PositionInvalid exception if the Iterator does not point at an element.

Resetting the Iteration

```
void reset ();
```

This operation resets the iteration to begin anew. The position of the Iterator is reset to the beginning of a Collection.

Testing for Completion of an Iteration

```
boolean more ();
```

This operation returns *true* if there are more elements to be accessed and *false* if there are not.

11.5 The Query Framework Model

The Query Framework interfaces provide an extensible framework for dealing with query. This is accomplished in two ways. First, by providing a standard, generic set of object interfaces for handling query. Second, by providing extensibility so that these object interfaces can be subtyped for further functionality.

The Query Framework interfaces define two levels of interfaces. The base level consists of *QueryEvaluator* and *QueryableCollection* interfaces and provides the minimal functionality for query. The advanced level consists of *QueryManager* and *Query* interfaces and provides an extensible functionality for dealing with all aspects of query.

11.5.1 Query Evaluators

A Query Evaluator is any object that supports the operation to evaluate a query. It can be a single object, an implicit collection of objects, or an explicit collection of objects (particularly a Queryable Collection, as discussed in Section 11.5.2 on page 11-20). An example of a Query Evaluator that manages implicit collections of persistent objects is a database system.

The result of a query evaluation can be anything. In most cases, it is a Queryable Collection, as illustrated in Figure 11-7. (The solid arrow represents operation invocation and the dotted arrows represent association.)

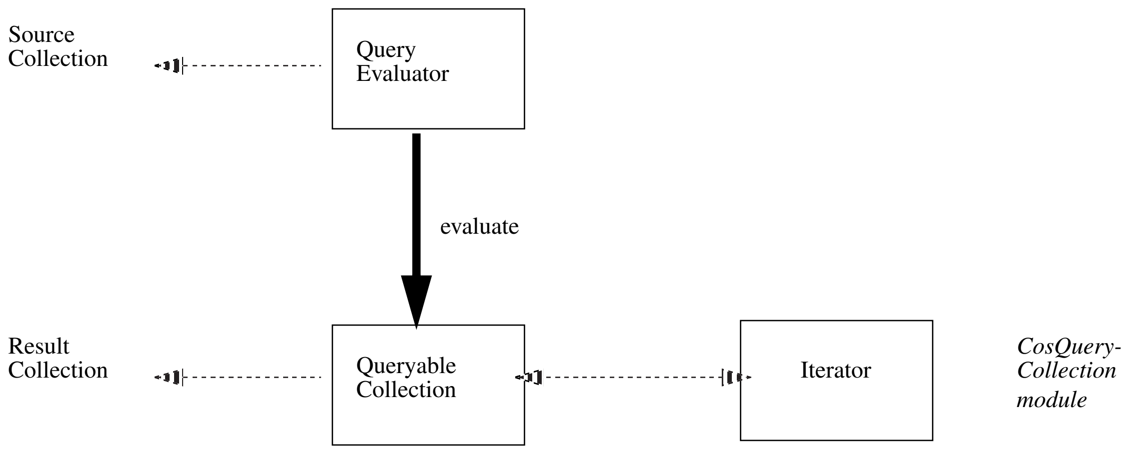


Figure 11-7 Query Evaluator and Queryable Collection

11.5.2 Queryable Collections

A Queryable Collection supports the *QueryEvaluator* interface and, therefore, can be used not only to represent the result of a query that consists of one or more objects, but also to define the scope to which further queries may be applied. An especially interesting kind of Queryable Collection is the type extent, whose member objects are instances of a certain object type.

A Queryable Collection evaluates a query by either invoking the evaluation operations on its member objects if they are Query Evaluators—or by evaluating the query predicate on the attributes and operations of its member objects if they are not—and by combining the results from such invocations and evaluations. As such, the query predicate must be a valid predicate for the Queryable Collection object and its member objects. If any one of its member objects is a Queryable Collection, the predicate (the applicable part, that is) must further be a valid predicate for its member objects, and so on. Therefore, the *QueryableCollection* interface provides a mechanism for nesting queries to an arbitrary number of levels.

11.5.3 Query Managers

A Query Manager is a more powerful form of Query Evaluator. It provides the operation to create Query objects. Working in tandem with a Query object, it manages the overall query processing and monitors the query execution. The *QueryManager* contains the universe of collections of objects over which queries can be specified. A specific query, as represented by a Query object, operates on a subset of this universe of collections.

The relationship between a Query object and its Query Manager is shown in Figure 11-8. (Dotted boxes represent logical entities; dotted arrows represent logical associations.)

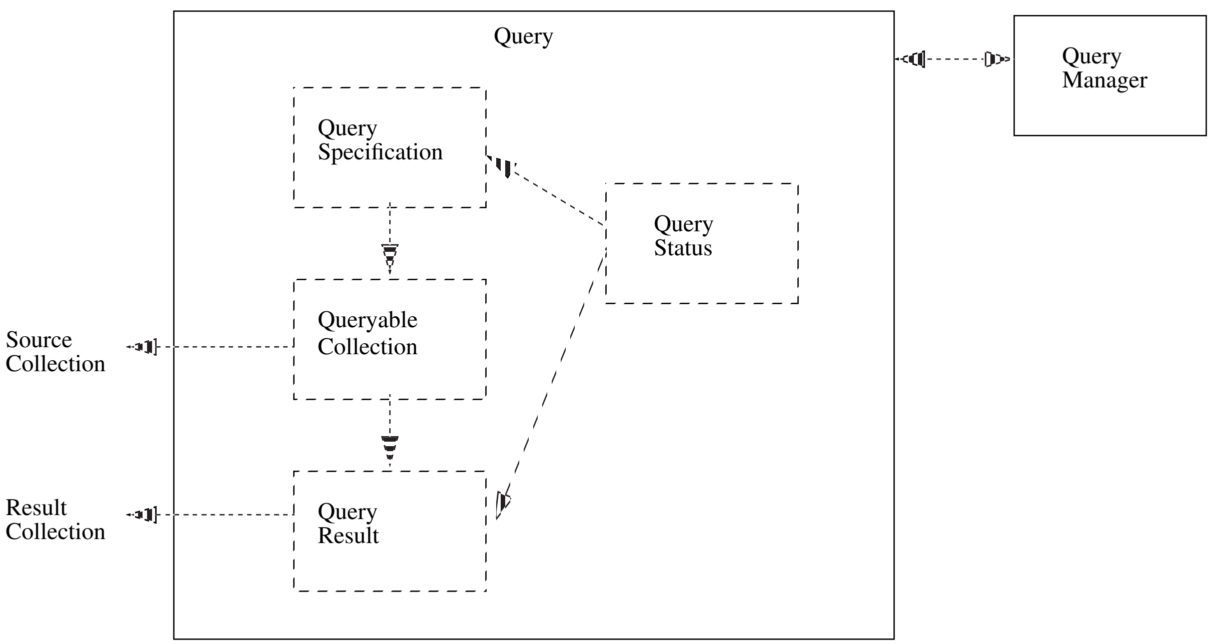


Figure 11-8 Query Manager and Query Object

11.5.4 Query Objects

A Query object represents a query and logically consists of the query specification, query status and query results. In addition, it contains the reference, either explicitly or implicitly through the Query Manager, to the queryable collection that defines its scope.

The Query object is responsible for composing and containing a query specification, including parameters. The query specification may be represented in the form of text, graphic, etc. A user may select a subset of the query specification to be executed in a query. This is particularly useful for query debugging. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for composing and selecting the query specification.

The Query object is responsible for maintaining the status information and log information regarding a query. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for displaying the status information.

The Query object also contains the results of a query. The *Query* interface is expected to be extended by vendors or users to provide the additional functionality for browsing query results. For example, successive results may be appended to previous results or replace them. A user may browse query results by specifying the version numbers, and so forth.

11.6 The CosQuery Module

The *CosQuery* module defines the query framework interfaces of the Query Service. In particular, it defines the following interfaces:

- *QueryLanguageType* interfaces to denote query language types.
- *QueryEvaluator* interface to represent query evaluators.
- *QueryableCollection* interface to denote collections which can serve as the result as well as the source of a query.
- *QueryManager* interface to create queries and perform query processing.
- *Query* interface to represent queries.

The *CosQuery* module is shown below.

```
module CosQuery {

    exception QueryInvalid {string why};
    exception QueryProcessingError {string why};
    exception QueryTypeInvalid {};

    enum QueryStatus {complete, incomplete};

    typedef CosQueryCollection::ParameterList ParameterList;
    typedef CORBA::InterfaceDef QLType;

    interface QueryLanguageType {};
    interface SQLQuery : QueryLanguageType {};
    interface SQL_92Query : SQLQuery {};
    interface OQL : QueryLanguageType {};
    interface OQLBasic : OQL {};
    interface OQL_93 : OQL {};
    interface OQL_93Basic : OQL_93, OQLBasic {};

    interface QueryEvaluator {
        readonly attribute sequence<QLType> ql_types;
        readonly attribute QLType default_ql_type;

        any evaluate (in string query, in QLType ql_type, in ParameterList params)
raises(QueryTypeInvalid, QueryInvalid, QueryProcessingError);
    };

    interface QueryableCollection : QueryEvaluator, CosQueryCollection::Collection {};

    interface QueryManager : QueryEvaluator {
        Query create (in string query, in QLType ql_type, in ParameterList params)
raises(QueryTypeInvalid, QueryInvalid);
    };

    interface Query {
```

```
            readonly attribute QueryManager query_mgr;

            void prepare (in ParameterList params) raises(QueryProcessingError);
            void execute (in ParameterList params) raises(QueryProcessingError);

            QueryStatus get_status ();
            any get_result ();
        };

    };
```

11.6.1 The QueryLanguageType Interfaces

The *QueryLanguageType* interfaces consist of seven interfaces that form the interface hierarchy illustrated in Figure 11-9.

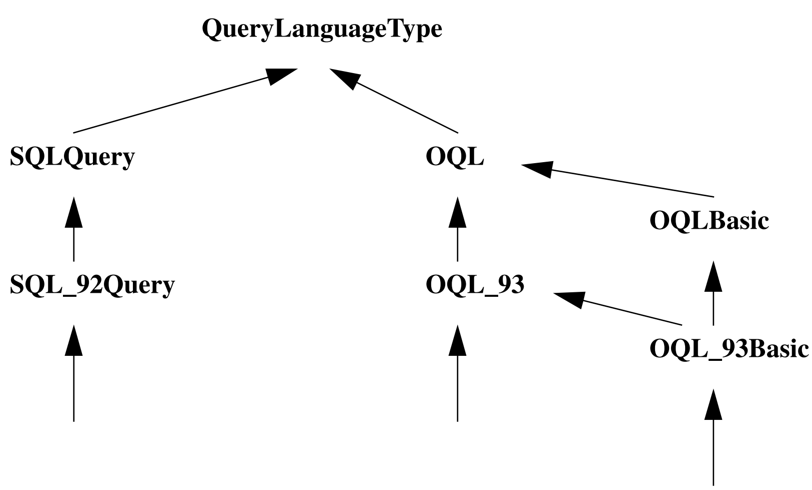


Figure 11-9 QueryLanguageType Interface Hierarchy

A Query Service provider is expected to use subtyping from SQL_92Query, OQL_93 or OQL_93Basic to denote the query language that it supports. For example, if a Query Service provider supports a query language, Object SQL, which complies with both SQL-92Query and OQL-93Basic, then its interface type, ObjectSQL, should be defined to be a subtype of SQL_92Query and OQL_93Basic:

```
interface ObjectSQL : SQL_92Query, OQL_93Basic {};
```

11.6.2 The QueryEvaluator Interface

The *QueryEvaluator* interface defines an operation for evaluating queries. It lets a client determine the query language types, and the default one, that it supports.

The result type of a query can be any. This is designed to accommodate generality. For most common queries, the results tend to be Collections (mostly of records or objects). For some specific queries, however, the result may be of any data type.

Determining the Supported Query Language Types

```
readonly attribute sequence<QLType> ql_types;
```

This attribute identifies the query language types supported by the *QueryEvaluator.*

Determining the Default Query Language Type

```
readonly attribute QLType default_ql_type;
```

This attribute identifies the default query language type supported by the *QueryEvaluator.*

Evaluating a Query

```
any evaluate (in string query, in QlType ql_type, in
ParameterList params) raises(QueryTypeInvalid,
QueryInvalid, QueryProcessingError);
```

This operation evaluates a query and performs required query processing. If the query language type is not specified, the default query language type is assumed.

The query language type specified must be supported by the QueryEvaluator. Otherwise, the QueryTypeInvalid exception is raised. If the query syntax or semantics is incorrect or if the input parameter list is incorrect, the QueryInvalid exception is raised. If any error is encountered during query processing, the QueryProcessingError exception is raised.

11.6.3 The QueryableCollection Interface

The QueryableCollection interface is a subtype of both the *QueryEvaluator* and *CosQueryCollection::Collection* interfaces. Any collection that supports this interface can be used to represent the result of a query that consists of one or more objects. It can also be used to define the scope to which further queries may be applied.

11.6.4 The QueryManager Interface

The *QueryManager* interface is a subtype of the *QueryEvaluator* interface. It defines an additional operation for creating Query objects. The QueryManager interface works in tandem with a Query object in managing the overall query processing and monitoring the query execution.

Creating a Query Object

```
Query create (in string query, in QlType ql_type, in
ParameterList params) raises(QueryTypeInvalid,
QueryInvalid);
```

This operation creates a Query object representing the input query. If the query language type is not specified, the default query language type is assumed.

The query language type specified must be supported by the QueryManager. Otherwise, the QueryTypeInvalid exception is raised. If the query syntax or semantics is incorrect or if the input parameter list is incorrect, the QueryInvalid exception is raised.

11.6.5 The Query Interface

The *Query* interface defines operations to:

- Prepare the query for execution
- Execute the query
- Determine the preparation and execution status of the query
- Obtain the result of the query

Determining the Associated Query Manager

```
readonly attribute QueryManager query_mgr;
```

This attribute identifies the *QueryManager* associated with the Query object.

Preparing the Query for Execution

```
void prepare (in ParameterList params) raises
(QueryProcessingError);
```

This operation performs the necessary processing, including optimization, on the query so that it is ready for execution. Query preparation may be carried out in cooperation with the associated *QueryManager.*

If the input parameter list is incorrect or if any error is encountered during query preparation, the QueryProcessingError exception is raised.

Executing the Query

```
void execute (in ParameterList params) raises
(QueryProcessingError);
```

This operation executes the query. If the query has not been prepared before, it will prepare the query first. Query execution may be carried out in cooperation with the associated *QueryManager.*

If the input parameter list is incorrect or if any error is encountered during query execution, the QueryProcessingError exception is raised.

Determining the Query Status

```
QueryStatus get_status ();
```

This operation returns the preparation and/or execution status of the query. This may be carried out in cooperation with the associated *QueryManager.*

Obtaining the Query Result

```
any get_result ();
```

This operation returns the result of the query.

11.7 References

1. American National Standard X3.135-1992, *Database Language - SQL*, January, 1993.
2. Object Management Group. CORBA: *Common Object Request Broker Architecture and Specification.* Published by the OMG, Framingham, MA. 1995.
3. Object Management Group. *Object Services RFP 4*, OMG Document Number 94.4.18, May, 1994.
4. Cattell, R.G.G. (ed), *The Object Database Standard: ODMG-93, v1.2,* Morgan Kaufmann Publishers, San Mateo, California. 1994.
5. Melton, Jim (ed), *SQL3 Part 2: Foundation*, ANSI X3H2-94-329, August, 1994.

Licensing Service Specification 12

12.1 Existing License Management Products

This section, "Background on Existing License Management Products," is for readers who are unfamiliar with the management of software licenses. It provides an overview of licensing and addresses issues that must be faced in developing and selecting a license management system.

Application suppliers need methods for controlling the access to and use of their products. In most cases, this is necessary to ensure fair compensation for use. The most common control method used by software suppliers is licensing, where the license can be provided through technical (software- or hardware-based) or contractual means. While contractual licensing is a viable option, it does not provide the same level of control as technical licensing, which uses hardware or software tools to control licensing. Therefore, application suppliers continue to require technical licensing methods to complement legal contracts.

Along with the expanding need for technical licensing, there are specific requirements for licensing that must change to reflect today's computing environments. Traditional licensing methods (nodelocked licensing and site licensing) evolved from computing environments of the past, specifically timesharing systems and stand-alone systems such as PCs and workstations. These older licensing methods are insufficient for current environments.

While today's computing environments provide significant advantages for application suppliers and end users, they also present opportunities. It is apparent that software and hardware resources can be managed on a network-wide basis for maximum efficiency. However, the resulting requirement for network-wide license sharing is less apparent. The traditional licensing methods (expensive site licensing and inflexible nodelocked licensing) do not complement today's fleixible and efficient computing environments.

Given these realities, sophisticated technical licensing tools are required. These licensing tools are important to all constituents in the market: application suppliers; hardware vendors; and application users. Software suppliers need a licensing tool to support their business and pricing models. Hardware vendors embed and offer the technology to support software developers and end users, and act themselves as application suppliers for their internally developed applications. End users interact with licensing technologies when they use, manage, and pay for software applications.

12.1.1 Business Policy

In the development and selection of software licensing systems, the licensing system must not impose its business practices on users. The software license is, in effect, a contract between suppliers and customers that establishes a business relationhip between them. Because a software licensing system plays an important role in regulating this contract, it must provide mechanisms to implement the flexible business practices that suppliers need to deal with a diverse customer base.

One danger in developing a licensing system is that it could reflect the business practices of the developing organization. This is sometimes the case when a licensing system is developed for internal use in a large organization and then offered for general use. A licensing system may work for one company, but will probably not address a wide range of business policies and practices. Often this problem manifests itself in subtle ways.

12.1.2 License Types

If not fully considered beforehand, it is possible to construct a software licensing scheme that forces the software suppliers and end users into a limited model of software licensing. If a licensing system offers only limited license types and/or offers few options for applying them, software suppliers are limited in the way they manage business relationships with their software customers.

Because software licensing touches many aspects of a relationship with a customer, including upgrades, support, enhancements, and follow-on purchases, a licensing system must provide a wide range of license options and many options for applying them. Software suppliers—not licensing system developers—must choose which licensing options they want to use.

The options allowed within various license types are also critical to ensure that application suppliers have all the capabilities necessary to establish the business relationship they desire with their customers. Capabilities such as allowing a grace period to provide unlicensed users access to the software for a limited period may be critical in retaining the goodwill of a large and influential customer. Other licensing features include selective user inclusion or exclusion lists; reserved licensing (to ensure that a license is always available to high-priority users); and multi-use rules that allow multiple use of an application with a single license. In addition, different license types can be used together in a single application. This ensures that the supplier, not the licensing system, determines business policies.

12.1.3 A History of License Types

Providing a wide-ranging portfolio of license types ensures that application suppliers are able to conduct business and arrange business policies as they deem appropriate.

Nodelocked licenses (which evolved from timesharing) allow a software product to be used at the single node for which the license was created. As the stand-alone workstation market grew, new licensing models were required. Major workstations users, such as insurance companies, banks, and industrial corporations, needing a more economical way to purchase software, demanded that application suppliers offer a business model that would provide unlimited use at a given site. That need gave rise to site licensing.

Site licensing often resulted in dissatisfaction of both suppliers and customers. Suppliers were asked to assess a price for usage they did not fully understand. They often felt they were being asked to discount their future revenue too deeply. Customers felt that the site license fees were excessive and made them pay for usage that might not occur in the future.

As networks of computers developed, system vendors began to introduce the notion of a concurrent use license. Concurrent use licenses define the number of users allowed to access an application at a given time. These licenses are allowed to “float” around the network, temporarily appropriated by users as applications are invoked, then returned to the license repository when an application is terminated. Concurrent use licensing allows end users to purchase licenses to match their usage and allows software providers to be compensated for use of their products. Additionally, end users can easily add more concurrent licenses as needed.

12.1.4 Asset Management

Licenses protect expensive corporate assets. Since licenses exist only as data they are harder to secure than a server or workstation, but every bit as important to control and manage. Control helps ensure that licenses are used in a manner which supports corporate goals, such as improving compliance with paper software license contracts and reducing exposure to legal action. This helps keep the corporation out of court and enhances its relationship with its software suppliers. Large corporate software purchasers want to be treated as equal partners with their suppliers; licensing makes this easier.

Managing both existing and new licenses maximizes their value. Old licenses might need to be redeployed as projects and budgets change. If the license administrator can keep track of software licenses, know which licenses are and are not being used, and can move them to where they are needed, corporate waste will be reduced and productivity improved.

Similarly, if a corporation has software usage metrics, it has a strong basis for understanding future needs. These metrics permit a corporation to purchase licenses in bulk at lower prices with the confidence that they are not over or under buying.

A corporation can also measure whether they have over or under purchased in the past. An important metric is the "shelfware" measure. How much software was purchased (perhaps as unused components of "suites" of software) that never leaves the user's bookcase? Reducing such waste is a major incentive for software customers to use automated software licensing and asset management.

12.1.5 License Usage Practices

Application suppliers can implement one or more of the license types in their software products. An application can be programmed to require multiple license types, to allow the supplier to sell the product in different ways to different customers.

An ideal licensing system should be transparent to end users. For example, a user might invoke an application, which makes calls to a licensing library. Then, the library function locates a server with a valid license. Assuming that a valid license is available and that person is authorized to use the license, a grant is returned to the application, allowing the program to execute, all completely transparent to the end user.

If no licenses are immediately available, the application developer can program the software to respond in a variety of ways. The application can automatically put the user on a wait queue, query the user as to the course of action to take, recommend that the user try again later, or grant permission to run anyway. (The developer can choose to grant permission to run without a license if, for example, there is a "grace period" instituted to allow for a smooth transition to a network licensing model.) If all licenses are temporarily checked out and users go on a wait queue, the next available license can be granted according to user priority settings defined at the end user site.

These choices and how they are implemented comprise the policy a software provider chooses as a business model. Policy can be further broken into two components: fixed and variable. Fixed components are coded into the application and determine things such as what license types are permitted; whether multiple use rules apply to the application; or if a grace period will be extended when a license is not available. Variable components are defined externally to the client application and include such things as external definition of the hours a product may be used, or an external list of people allowed to use it. Either list may be producer- or end user-created.

12.1.6 Scalability

Some networks are small, consisting of just a few nodes, while others grow to thousands of machines. Typically, large user communities on large networks demand licensed applications from many different vendors. A licensing system and its runtime environment must, therefore, scale well to the network and all its software.

12.1.7 Reliability

Sometimes, an application obtains a license from a license server and never returns it. A licensing system must be designed to prevent licenses from being stranded and to prevent other client-server breakdowns.

Breakdowns occur for several reasons. The application or server could abort, or the network could become partitioned between the application and server. These situations could arise unintentionally or maliciously (for example, in an attempt to gain unauthorized use of an application). Any design must make careful trade-offs between license availability and security enforcement. All designs require a scheme to detect breakdowns.

Generally, there are two detection methods: continuous detection or occasional check-in. Continuous detection methods ensure that while a license is in use by an application, the application and server are both continuously aware of each other's existence and are immediately notified of a breakdown. These schemes are typically implemented by using a connection protocol such as a port. The main advantages of a continuous scheme are its directness, immediacy, and simplicity. The main disadvantage is its negative impact on network performance. If a redundant server high availability model is employed, then continuous connections need to be maintained between the application and each of the servers, thereby multiplying network traffic.

Occasional detection provides a method for the application to check in with the server periodically before some time out has occurred. The breakdown is identified either by the server (if the time out occurs), or by the application (if the check-in is unsuccessful). This method is very scalable and has a negligible impact on performance. The application supplier should be able to adjust the time-out to allow trade-offs between higher security and higher availability.

Additionally, the occasional detection model is very tolerant of momentary interruptions on the network. Continuous detection is not tolerant of such interruptions. Lost connections between the client and the server in a system using continuous detection causes a breakdown or program termination.

Application suppliers will want to determine for themselves which action to take in the case of a client-server breakdown. Some may want more strict enforcement and choose to terminate an application; others might choose to display a polite message and allow the application to continue.

12.1.8 Legacy Applications

Managing a business relationship with a minimum of disruption includes the ability to accommodate existing customer applications within the scope of the licensing system. This must be done without requiring access to or modification of the application's source code, as the apparently simple solution of modifying source code may not always be available. Consider the personal computer, for which there are tens of thousands of small and inexpensive applications. Modifying the sources of all these applications would be an economically unacceptable approach even if the source code were available.

Software suppliers are eagerly awaiting an integrated licensing technology that will take existing “shrink-wrapped” applications and enhance them to function in a licensed environment. It may not be possible to provide a security fence as high as a source-modified application, but the level of license security could be made commensurate with the value of the application and well beyond the economic justification for attempting to defeat the security.

12.1.9 Security

Until recently, licensing systems were required to enforce only simple, single-system application use. Security infractions caused few implications. Today, security requirements must be designed to operate in more complex networks.

The distributed computing networks in use today are designed for easy resource sharing; demand more complex licensing models (presenting new security challenges); and must support mass distribution of software (on compact disk, for example). A supplier's ability to ship trial copies of applications relies heavily on the security of the licensing system to ensure that prospective customers do not transgress the intended use permissions. An application supplier must also rely on the licensing system's security when it ships a complete set of applications to its entire installed base: the licensing system must ensure that only the purchased applications are used.

Each application supplier has a different security need. Each will want to choose from a spectrum of trade-offs, such as security versus availability, and effect of breach versus development effort. A licensing system should not dictate one particular level of security, but should allow application suppliers to choose the security level appropriate for their business needs.

12.1.10 Client/Server Authentication

A secure licensing system should address the possibility of someone attempting to create an impostor license server (an imposter server always grants licenses). Without security, an impostor could be established by eavesdropping on valid client-server communication and then mimicking the license grant protocol. Impostor clients should also be addressed, since a successful impostor client could disrupt legitimate license activities by artificially returning a license to the license server when it is actually still in use, thereby making the returned license available for other users.

12.1.11 Example: Application Acquiring and Releasing a Concurrent License

This section contains an example of how an application might interact with one of the various license management products that exist today.

In a system that uses concurrenct licensing, end users at their workstations and PCs see no change in their normal working routine. They start applications as they normally would. The application has calls to the license library that transparently go over the network to request a license for the application. Using transport-specific naming and location facilities, a server holding a valid license is located and a “yes” is returned to the requesting application.

The application need not be downloaded over the network to the workstation each time the application is invoked. The application, once loaded, remains at the workstation as it normally would. Only a request for a license and a return grant go over the network, thereby providing a rapid response time that is virtually unnoticed by users.

When end users close an application, the license is “returned” to the server. The server then can make this license available for other requests as they come in.

Administration and reporting tools act as clients to the license server, tapping into server databases and log files to access the stored information. The license servers, though implemented as multiple physical servers, operate as a single repository managing all license activity for the network. This single, “logical” server handles licenses for any number of vendors, for any number of products, with any number of product versions. The server also handles any number of clients making requests for its facilities, thereby automatically scaling to accommodate increases in the number of users, machines, applications, and licenses.

12.2 Service Description

Licensing Service terms are defined in Appendix A.

12.2.1 Overview

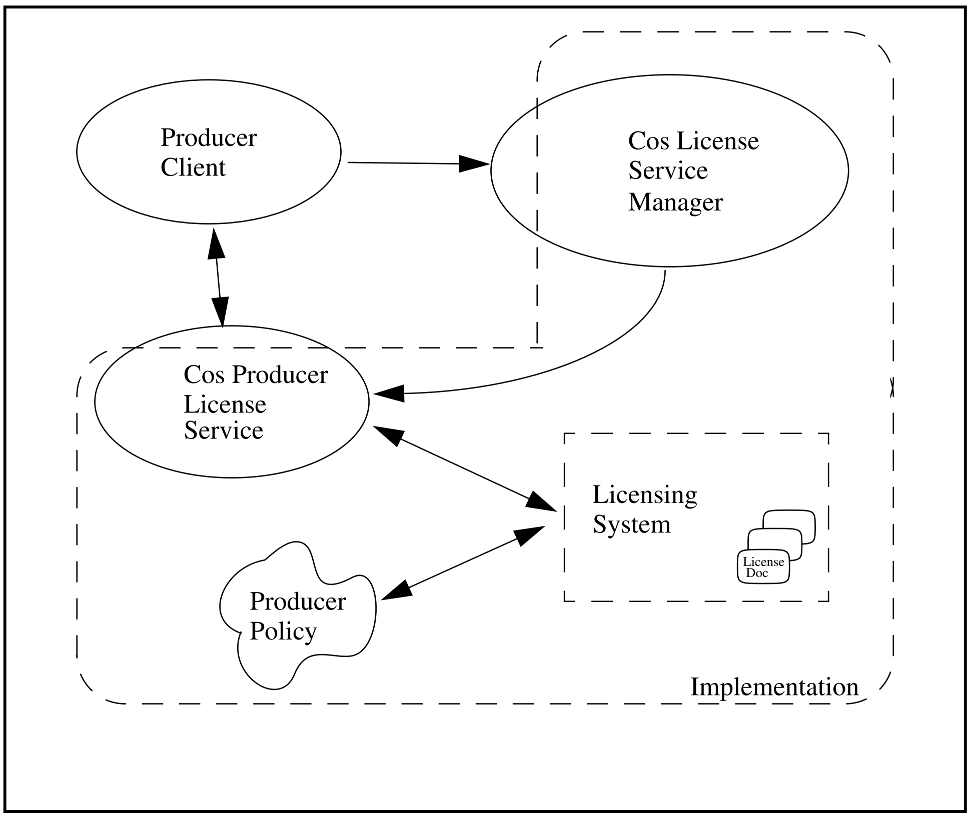


Figure 12-1 Licensing Service Relationships

The Licensing Service provides a mechanism for producers to control the use of their intellectual property in a manner determined by their business and customer needs. In Figure 12-1, the Licensing Service Manager, Producer Licensing Service, and the Licensing System are shown as three distinct objects. Implementations of the Licensing Service may differ. The dotted line indicates components that depend on the implementation design and are addressed in terms of an example solution. Components outside the dotted line are addressed in this chapter.

12.2.2 Key Components of a Licensing System

License Attributes

To implement controls, the Licensing Service needs a set of fundamental attributes. A license can be thought of as having three dimensions of attributes:

Time includes, but is not limited to the attribute of Expiration/Duration. All licenses should be able to have start/duration and expiration dates.

Value Mapping includes, but is not limited to, the following attributes:

- A *unit* is a quantity that can be used by policy mechanisms.
- *Allocative*. Use of an license with an allocative attribute removes it from the pool of available allocative licenses for a given product until it is returned. This is traditionally known as concurrent use licensing.
- *Consumptive*. Use of a license with a consumptive attribute permanently records its use. This can be used to provide metering capability. It can also be used to implement a "grace period" via the use of overflow licenses when the maximum number of allowed concurrent licenses has been met.

Consumer includes, but is not limited to, the following attributes:

- *Assignment or Reservation.* All licenses should be able to be assigned to or reserved for a specific entity or collection of entities. The definition of what an entity may be is implementation-specific. One example is where an entity is defined to be a specific user and a collection of entities is a specific organization comprised of a collection of specific users. Other examples of what an entity might be include a specific machine or collection of machines, a specific system resource or resource collection, such as printers and adapters.

Licensing Policy

The Licensing Service allows the license attributes to be combined and derived from to form any policy deemed necessary. This allows the producer and, where appropriate, the end user administrators to control product use to fit their business environmenst.

The following derived attributes are representative examples of those that can be used for a flexible policy implementation:

- Time windows
- Value
- Use by a collection of related objects
- Postage meter
- Gas meter

Time Windows

It may be necessary for some policies to constrain the time periods within which a particular license unit may be used. A time window attribute can be derived from the expiration/duration attribute.

Value

A Producer can define, as part of their Producer Policy, the mapping between actual use of their intellectual property and the way license units are associated with that use in the Licensing System. A simplistic example might be where a single unit of control represents a single active implementation of a given object with no constraints on the number of instances. A more complex example may be where the number of units of control required may be calculated to satisfy a combination of requirements: a specific machine size where an implementation is active, how many instances, and how many method activations are allowed in parallel.

Use By a Collection of Related Objects

The definition of granularity is very broad. In an OMA-compliant system, the Licensing Service will allow control from the fine grain of a method activation to the coarse grain control of a suite of objects acting together in a relationship to represent an application. The relationship may be defined with the Relationship Service, a future Collection Service or any other Service providing relational capability for objects. The Producer Policy can discover all theseobjects according to the implementation.

Postage Meter

Derived from consumptive, use of a license with a postage meter attribute permanently removes it from the pool of available licenses. The total number of licenses is never less than zero (0) for any product.

Gas Meter

Derived from consumptive, use of a license with a gas meter attribute adds to the pool of consumed licenses. The total number of licenses is initialized to 0.

Examples of how these attributes can be used in license policy are as follows:

- An end user administrator could be empowered by the Licensing Service to combine assignment and time constraints on installed license units to constrain the use of certain products to a set of individuals outside of the normal work week.
- A producer could provide a personal use license by combining an allocative attribute with an assignment attribute to an individual with a unit attribute of 1.
- A producer could enhance the previous example by allowing end user administrators to reassign the license to a particular group.

Interfaces Isolated From Business Policies

The Licensing Service interfaces are isolated from policy issues. The client interface only delivers notification that a producer wants some or all of the producer's intellectual property to be controlled reliably and securely. Once the notification is made, the Licensing Service can identify the appropriate policy.

For example, consider a producer who wants to restrict the activation of a particular method to a certain simultaneous number of users. The producer need only tell the Licensing Service interface to indicate that a method has been activated and who activated it. When the method activation is complete, an indication must be sent that the use is done. The LS can then, in an implementation-specific way, determine if a producer-defined limit has been met. The Licensing Service can notify the object, telling it what to do if a producer policy is activated from overuse or another condition. The Producer can still override a generic policy with an alternate behavior for a particular Producer Client , since policy responses are inside the Client implementation.

A Producer Policy implementation requires the use of other object services such as the Relationship and Property Services. As other services are defined, producer policy implementations will broaden to use them. The producer client might change to address any new producer policy, but the underlying Licensing Service interface will not require change. These services can be used to find out about objects outside of the objects themselves.

For example, consider the Relationship Service. If producers choose to license a particular set of their objects that are related in a manner defined by the relationship service, the producer policy implementation can obtain relationship information using the relationship service. The objects involved need to have no special knowledge about their relationships to one another other than that required to conspire together in the relation to achieve their desired functionality. Mechanisms provided to support this by an particular implementation will vary. One implementation may choose to support this using a document style of policy delivery, others may support producer policy object implementation. This can not be defined or restricted by the Licensing Service client interface.

A mechanism for license document delivery is not defined in the Licensing Service: it is implementation-specific.

12.2.3 Licensing in the CORBA Environment

Licensing in the CORBA world faces many issues. The provision of services by objects in the ORB environment must allow for service producers to control use of their intellectual property according to their business models.

Constraint of use must range from strict control to benign monitoring of intellectual property. Strict control might allow only a specific number or combination of services to be used. Benign monitoring mechanisms might allow service use without constraint, but would track usage for later examination.

If producers require strict control, they will also require assurance that the information provided by their licensing mechanism is secure. It would be pointless to choose strict control if it were a trivial matter to replace some component within the ORB which nullified strict control enforcement without the producers' services being aware of it. The level of trust in the Licensing Service must meet the producer's chosen enforcement policy. For example, suppose a producer has selected a policy that allows use of his object service by an end user without constraint, but the policy requires the Licensing Service to log all service usage so a monthly post-facto charge can be made for use of the service. This capability is of limited use if the Licensing Service's logging mechanism allows end users to illicitly modify the logs to show low usage.

To enable usage control, there must be a mechanism that provides the end user with appropriate authorization. This authorization is usually conveyed as a text string that can be thought of as a License Document. The size of this document may vary from a few tens of characters to a few thousand characters depending on the functionality provided by the underlying Licensing Service. The content of the document must be protected by an implementation-specific mechanism.

To support a wide variety of business models, producers require usage constraint policies (producer policies) that can vary for end user conditions. For example, a producer might deliver a demonstration of a client service that allows unlimited use of the service during the demonstration period, but upon purchase requires a strict usage enforcement policy. The enforcement policy may need to be varied depending on customer needs. A large customer may negotiate a post-sale period where analysis of use is supported by benign monitoring and later moved to strict enforcement. Interfaces to the Licensing Service allow this and many other varieties of usage controls without requiring changes to the producer's fundamental product.

The ability for an end user to apply constraints beyond those specified by a producer is a well-recognized benefit to the end user. The capability in this area will vary across implementations of the Licensing Service.

Because we live in a dynamic economic environment, a producer's policies must be easily changed. The best approach for a Licensing Service specification is to separate the "I want to be controlled!" requirements of the application or service from the "how am I to be controlled?" requirements of the policy that have to deal with all of the exceptions and producer business practices. This separation enables a producer to choose a Licensing Service implementation based on considerations of how well a specific Licensing Service supports the producer's business practices, as instantiated by the producer policies.

The interface to the Licensing Service accomplishes this by allowing the controlled applications or services to notify the Licensing Service of its wish to be controlled specifying how the enforcement is to be performed.

Administration and policy issues are not addressed in detail by the Licensing Service interface; instead, they are left to implementors. End users need to control their own interface and reporting capabilities. The ability of the underlying Licensing Service to generate management reports, both of historical and snapshot-of-time usage, will vary widely depending on the implementation. The administrative interfaces for the Licensing Services include command line only, GUI only, and combined GUI and command line. An administrative interface would affect the ability of end users to manage their environments as they choose, so it is not defined by the Licensing Service.

12.2.4 Design Principles

The design of the Licensing Service interface satisfies the following principles:

Neutrality. The Licensing Service should not introduce any constraint on the way a Producer can use the interfaces because of some underlying dependency on the LS implementation. Producers need to be able to choose Licensing Service implementations that allow them to deliver their products in a manner best suited to the individual Producer's business needs without requirements on the way the interface is used. It is expected that LS implementations will allow many Producer Client objects to reference a single instance of the associated *ProducerSpecificLicenseService* interface to reduce the overhead of object creation.

Extensibility. The Licensing Service allows for extensions to support styles of Producer Policies that are not currently obvious. The Licensing Service provides extensibility in its object reference in the returned Action structure in the check-use operation. This allows implementation-specific extensions to the notification mechanism. The interface can also be extended by adding additional arguments and/or operations; for example, in support of the Security Service.

Security. The Licensing Service provides a mechanism such that a degree of trust can be established between the users of the interface (the Producers) and the underlying license management system. This is different from a typical secure environment since, the Producer does not usually trust the end user or the end user security environment. A mechanism is provided to allow the Producer to authenticate, in real time, that the underlying license management system is a legitimate provider of the Licensing Service. End user administration can not circumvent this authenticating mechanism.

Performance. Implementations of the Licensing Service may choose to optimize performance by the manner in which Producer Specific Licensing Service objects are managed. For example, an implementation could choose to allow multiple copies of a Producer Specific Licensing Service to distribute client operations.

The Licensing Service mechanisms must allow both synchronous and asynchronous messages so a producer can decide what is best for its application. For example, a very short duration method activation may well be best suited, for performance reasons, to using asynchronous meechanisms. On the other hand, if producers want to be extremely strict, they might choose synchronous messages to prevent misuse and accept the resulting loss of performance.

The Licensing Service provides mechanisms so that an application using the Licensing Service cannot accidentally orphan a license by acquiring an allocative style of license and never releasing when an application fails. Current mechanisms include connection-oriented, client-server communications; client-server heartbeat mechanisms; and server-based, client status query mechanisms. Keep in mind that the mechanism chosen may place a performance burden on the producer client.

12.2.5 Licensing Service Interfaces

The Licensing Service defines the interface between the Producer Client and the Licensing Service Manager (*LicenseServiceManager* interface) and the interface between the Producer Client and the Producer Licensing Service (*ProducerSpecificLicenseService* interface). The interfaces enable Producers to control use of their intellectual property in any manner they deem appropriate for their business model. The isolation of policy from the Licensing Service interfaces enhances Producer flexibility. The interfaces for administration, policy creation, and license document creation are not addressed, because they are implementation-specific.

The *LicenseServiceManager* interface provides a mechanism for the Producer to locate an object supporting the second interface, *ProducerSpecificLicenseService*. All of the operations required to constrain use of producers' intellectual property are supported by the second interface. This design allows the implementors of Licensing Services to make trade-offs such as those between client performance, licensing system performance, and ease of administration.

Once a Producer Client implementation has obtained a *ProducerSpecificLicenseService* object reference, the three operations (start_use, end_use, check_use) can be performed on this interface within the Client where the Producer deems it correct. The information returned from these operations provides the basis for the Producer to enforce its chosen usage constraint policy.

Interfaces are Mandatory

All the interfaces are mandatory for all implementations. Optional arguments exist in the *LicenseServiceManager* interface. For the *check_use* operation the ProducerNotification component of the returned Action can be a nil object reference indicating that the implementation does not support this kind of notification mechanism. In the *start_use* operation the call_back argument can be a nil object reference indicating that the Producer client implementation is not using event services and is designed to operate in a poll only mode. The properties argument to *start_use*, *check_use*, and *end_use* can be nil.

Constraints on Object Behavior

The Licensing Service interface assumes the provision of an Event Service (the Event Service is specified in Chapter 4 of this manual. If an Event Service implementation supports true asynchronous events—where delivery of an event can interrupt an object's task to invoke the push operation—then the Producer Client implementation must manage its internal state in a re-entrant world.

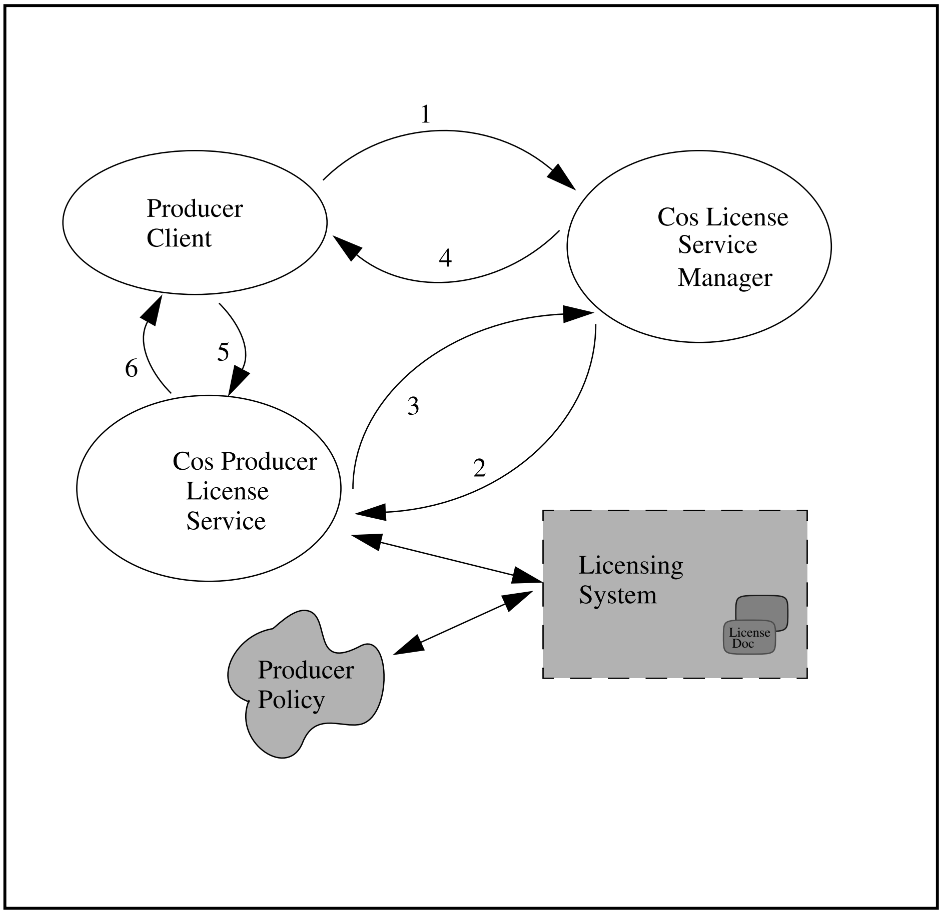


Figure 12-2 Licensing Service Instance Diagram

In Figure 12-2, the Producer Client performs the operation obtain_producer_specific_license_service on the *LicenseServiceManager* interface (Step 1). The Licensing Service Manager implementation creates an object (Steps 2 and 3) or locates an object reference to an object who has an interface *ProducerSpecificLicenseService* and who is capable of responding to the particular producer challenge. It then returns the reference to the Producer Client (Step 4). The producer client now uses the reference to perform the operations start_use, check_use, and end_use (Steps 5 & 6). In implementations that support true asynchronous events, the *ProducerSpecificLicenseService* object can asynchronously perform the push operation using the reference to the interface in the Producer Client provided as one of the arguments to the start_use operation in a previous step (in Step 5).

12.2.6 Licensing Event Trace Diagram

Figure 12-3 on page 12-16 represents the flow of events through Producer Client objects and a Licensing Service implementation. The steps below are illustrated in the diagram.

1. Producer Client gets an object reference to the Producer Specific Licensing Service.

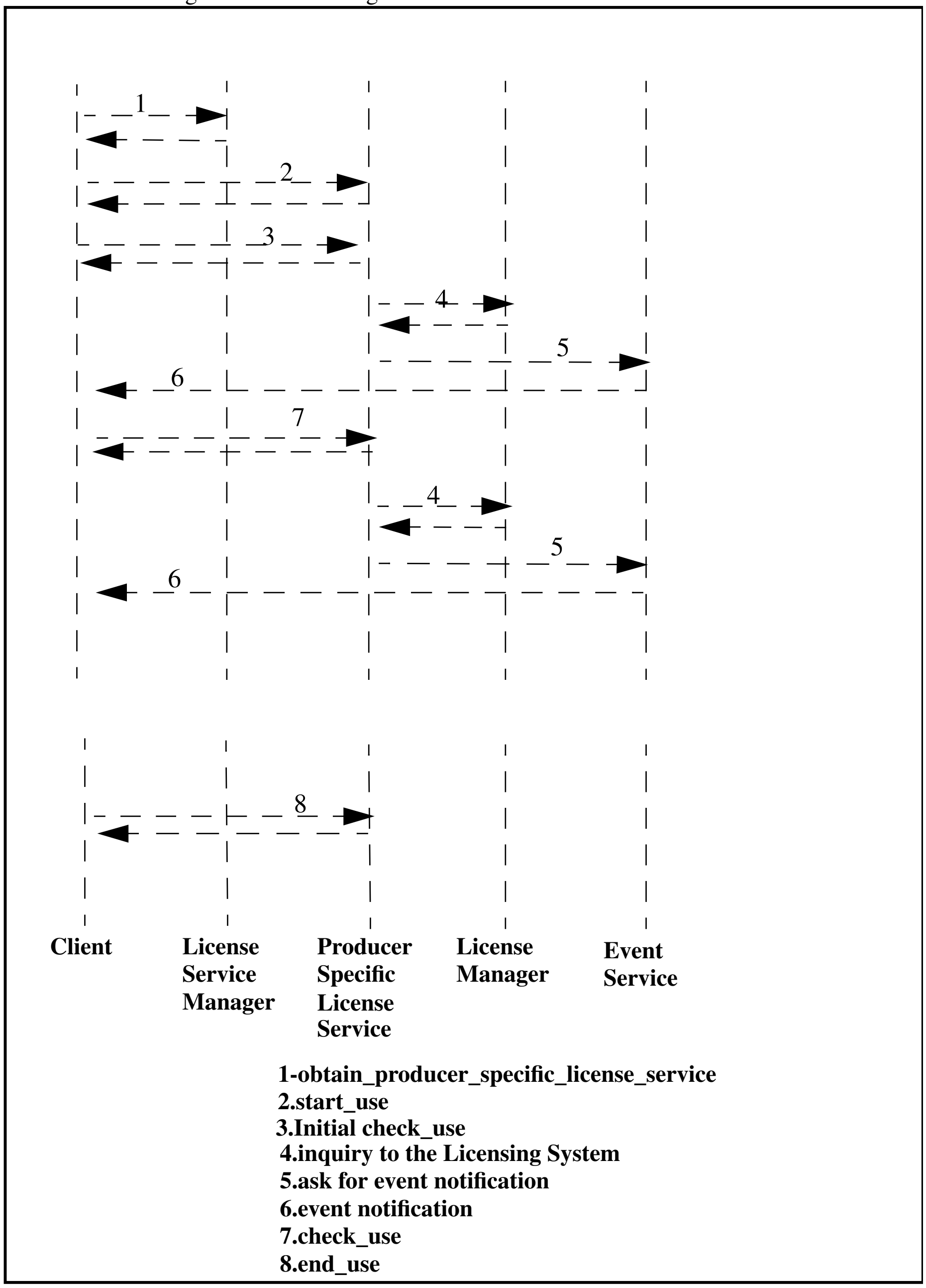
2. Producer Client determines that usage control is required and performs the `start_use` operation.
3. Producer Client does an intial `check_use` call to retrieve the initial recommended_check_interval.
4. Producer Specific Licensing Service instance interprets policy and interacts with the Licensing System as necessary.
5. If asynchronous events are supported, the Producer Specific Licensing Service asks for event notification to the particular Producer Client at an interval determined by Policy.(See page 35 for information on asynchronous events).
6. Event Service delivers the event to the Producer Client.
7. Producer Client responds to the event by performing the `check_use` operation.

Steps 4,5,6,7 are repeated until the Producer Client instance indicates that usage control is no longer necessary.

8. Producer Client performs the end_use operation when usage control is to be terminated.

If asynchronous events are not supported, the Client implementation will need to "poll" the Producer Specific Licensing Service with the `check_use` operation at an interval defined by the check_interval argument to the `check_use` operation. To initially retrieve this check_interval value, the Client will need to invoke a `check_use` immediately after the `start_use` call.

Figure 12-3 Licensing Event Trace Diagram



12.3 The CosLicensing Module

The CosLicensing module is a collection of interfaces that together define the Licensing Service. The module contains two interfaces:

The *LicenseServiceManager* interface consisting of the following operation:

- obtain_producer_specific_license_service

The *ProducerSpecificLicenseService* interface consisting of the following operations:

- start_use
- check_use
- end_use

This section describes the *LicenseServiceManager* and *ProducerSpecificLicenseService* interfaces and their operations.

The CosLicensing module is shown below. Note that this module definition uses some definitions from the *CosEventComm* module (in the Event Service) and the *CosProperty-Service* module (in the Property Service).

```
#include "CosEventComm.idl"
#include "CosPropertyService.idl"

Module CosLicensingManager {
           exception InvalidProducer{};
     exception InvalidParameter{};
     exception ComponentNotRegistered{};

    typedef Object ProducerSpecificNotification;

        enum ActionRequired { continue, terminate};

enum Answer { yes, no };

    struct Action {
          ActionRequired action ;
          Answer notification_required ;
          Answer wait_for_user_confirmation_after_notification ;
          unsigned long notification_duration;
          ProducerSpecificNotification producer_notification;
          string notification_text;
};

    struct ChallengeData {
          unsigned long challenge_index;
          unsigned long random_number;
          string digest;
    };
```

Figure 12-4 CosLicensingManager Module

```
#include "CosEventComm.idl"
    struct Challenge {
          enum challenge_protocol { default, producer_defined };
          unsigned long challenge_data_size;
          any challenge_data;
    };

    typedef any LicenseHandle;

        interface ProducerSpecificLicenseService {

 readonly attribute   string producer_contact_info
 readonly attribute   string
producer_specific_license_service_info

        LicenseHandle start_use (
           in Principle principle,
                in string component_name,
           in string component_version,
           inProperty::PropertySet license_use_context,
           CosEventComm::PushConsumer call_back,
           inout Challenge challenge
                          )

        raises ( InvalidParameter, ComponentNotRegistered);

        void check_use (
            in LicenseHandle handle,
                       in Property::PropertySet
license_use_context,
                out unsigned long recommended_check_interval,
                out Action action_to_be_taken,
                inout Challenge challenge
                                  )

                raises ( InvalidParameter );

        void end_use (
                in LicenseHandle handle,
            Property::PropertySet license_use_context,
                inout Challenge challenge
                      )

          raises ( InvalidParameter );

    };

    interface LicenseServiceManager {
      ProducerSpecificLicenseService

obtain_producer_specific_license_service (
                         in string producer_name,
                         inout Challenge challenge
                         )
```

Figure 12-4 CosLicensingManager Module

```
#include "CosEventComm.idl"

                            raises ( InvalidProducer, InvalidParameter };
        };
};
```

Figure 12-4 CosLicensingManager Module

Exception Raised	Description
InvalidProducer	Indicates that the producer argument is not correct or that an appropriate producer cannot be found.
InvalidParameter	Indicates that one of the parameters is invalid. No additional detail is provided in this document since this will include a failed challenge. Additional information could assist if someone wanted to make a deliberate attempt to work out the challenge of a producer.
ComponentNotRegistered	Indicates that the specific component has not been registered with the Licensing System.

Table 12-1 Exceptions Raised by Licensing Service Operations

12.3.1 LicenseServiceManager Interface

The *LicenseServiceManager* interface defines a single operation: obtaining the producer specific Licensing Service object.

The *LicenseServiceManager* interface allows a producer to control the use of their intellectual property. The `obtain_producer_specific_license_service` operation returns an object reference that supports the *ProducerSpecificLicenseService* interface. This operation is protected by the use of a producer challenge.

It is likely that implementations of the *License ServiceManager* will make use of other Object Services, such as Life Cycle, to create a producer-specific instance of the Licensing Service. The Life Cycle Service is not used directly in order to allow the service implementation to cache object references for performance reasons. Requiring instance creation on every use of the `obtain_producer_specific_license_service` operation is not desirable, but can be allowed in a particular implementation.

The operation `obtain_producer_specific_license_service` raises the InvalidProducer and InvalidParameter exceptions.

12.3.2 ProducerSpecificLicenseService Interface

The *ProducerSpecificLicenseService* interface defines three operations: notification that a product has started to be used, notification that a product is still in use, and notification that a product has finished being used.

Any object that possesses an object reference that supports the *ProducerSpecific LicenseService* interface and is capable of satisfying the challenge for that particular instance of the *ProducerSpecificLicenseService* interface can perform the following operations:

- The `start_use` operation which allows producers to notify the License Service that some aspect of their product has started to be used and is to be controlled by the service.
- The `check_use` operation which allows the producers to notify the Licensing Service that some aspect of their product that previously notified the service using a `start_use` operation is still in use.
- The `end_use` operation which allows the producers to notify the Licensing Service that an aspect of their product, previouslynotified to the service in the `start_use` operation, has completed its use.

All of the previously listed operations are protected by a challenge mechanism to allow a producer to be satisfied that the instance of the Licensing ServiceManager is a legitimate one to control the producer's intellectual property.

The attribute producer_contact_info may be used to provide information that can be displayed to an end user. The attribute producer_specific_license_service_info can be used, if necessary, for a Producer Client to alter the way it interacts with different *ProducerSpecificLicenseService* objects. These attributes are defined at creation of the *ProducerSpecificLicenseService* instance and do not change during the instance's life.

The start_use, check_use and end_use capture and propagate information about the user's runtime context to the Licensing Service via the *license_use_context* parameter. This information will typically include the user's name, their node's name, network address, local time, and so on. This information can then be used by the License System for a variety of purposes:

- In an access control mechanism to determine whether or not to allow the user to continue.
- In a private, possibly secure, usage logging mechanism.
- To provide data for peripheral management functions, such as triggering an e-mail message to the network administrator when resources run out.

The operations `start_use, check_use`, and `end_use` raise the InvalidParameter exception.

The `action_to_be_taken` output parameter in the `check_use` operation is used to give the ProducerClient information on actions to be taken as a result of its request to be active or running. The following describes the Action structure in more detail. Note that only the *action* field must be specified. All other fields can return a value of NULL in which case behavior is determined by the coded policy defined within the ProducerClient implementation.

- *action* : This field indicates if the ProducerClient should continue or terminate its processing depending on whether the requested license is available from the Licensing System.
- *notification_required* : Indicates whether or not the ProducerClient needs to prompt the local user with a message indicating the results of the licensing request.
- *wait_for_user_confirmation_after_notification* : Indicates whether the ProducerClient needs to wait for a confirmation before continuing its processing. This is applicable only if a notification has been requested.
- *notification_duration* : If the user notification is required without confirmation, this indicates how long the ProducerClient needs to wait before continuing with its processing.
- *producer_notification* : This provides a reference to an object used by a Licensing System to return implementation specific results and control information to the ProducerClient. For example, producer policy instructions can be part of this object interface. It could also communicate the expiration date and time.
- *notification_text* : This provides the text to be communicated to the local user if required.

The `check_use` operation thus collects into a single client *action* the ability to address the following requirements:

- Give the capability to the producer client to get both the results from and the actions to be performed following a request for permission to be active and/or running.
- Give the capability to the producer client to periodically verify the right to be active and/or running in the case of 'time dependent' licensing policy (for example, time based consumable licenses, expiration times, and so forth). The recommended_check_interval is the parameter strictly tied to this verification.
- Give the capability to both the producer client and the Licensing Service implementation to detect the following *unexpected* conditions and then either release the related active license and/or stop the usage accounting:
  - Abnormal termination of either the producer client or the Licensing Service.
  - Unrecoverable breakdown in communication between the Producer Client and the Licensing Service.
  - The indirect detection of these conditions is performed by forcing the producer client to issue a check request within the check interval.

The check request concept is left to the specific Licensing System implementations. However, that does not prevent the Licensing Service from using the check operation as the heartbeat mechanism. The heartbeat mechanism is a general purpose mechanism required inside a client/server based application to determine if the other end is still active. Some applications dedicate a specific process or task to this purpose and rely on event detection, others use a polling mechanism, others use system notification exits, and so on. Furthermore, because of the different concepts, the polling and exits could not be fully satisfied by a single checking rate.

12.4 References

Object Management Group. *Object Services RFP 4*, OMG Document Number 94.4.18, May, 1994.

Appendix A Licensing Service Glossary

License Document: Represents the fundamental element of control. It provides a secure delivery vehicle describing such things as how many copies of the intellectual property are allowed, how long each copy may be used, and other elements of how producers wish to constrain usage of their intellectual property.

Licensing Service: The general term for the complete service, it consists of three components: Producer Client; Producer Licensing Service; and Licensing Service Manager.

Licensing Service Manager: The Common Object Service Licensing Service Manager is responsible for managing and creating the Producer Licensing Service objects.

License Unit: License documents may contain the concept of license units that are interpreted in a producer-specific manner by the producer policy. A typical example of a license unit could be one where a single unit is to represent a single concurrent use of a producer's intellectual property by an individual user. The term license can be used to refer to the smallest indivisible quantity of license units that a given Licensing System implementation supports.

Licensing System: The implementation-specific component that provides fundamental usage control that, in conjunction with the Producer Licensing Service, provides sophisticated producer policies. The Licensing System is responsible for securely managing the fundamental units of control - the License Documents for all Producers.

Producer: The company or individual who owns the intellectual property that requires usage control.

Producer Client: Any object, or component of an object, that wants to have its usage controlled or metered via a Licensing Service.

Producer Policy: A Producer Policy is a collection of data that describes the detailed terms and conditions, or business policies, which govern control and monitoring of a producer's intellectual property wherever the property can be used. The implementation of producers' policies is very specific to the Producer's selection of a Licensing System. There are two components to business policy implementation in a licensing system. One component is contained in the License Document and includes fundamental things like expiration date and quantities. The other component, the Producer Policy, includes the broader aspects of business policy and may be derived from the License Document. As an example of the broader issues that require Policy, the Producer Policy deals with all possible licensing exceptions such as when no license is found.

Producer-Specific Licensing Service: A producer-specific implementation that interacts with and selects the particular Licensing System and Policy used by a specific Producer to control the Producer's intellectual property. In this chapter, the Producer-Specific Licensing Service is is also referred to as the Producer Licensing Service.

Appendix B Use of Other Services

This appendix describes the relationship between the Licensing Service and these Object Services: Property; Relationship; and Security.

B.1 Property Service

The properties argument to the `start_use, check_use` and `end_use` operations enables implementations to choose between using the Property Service or providing name value pairs directly to the Licensing Service. This decision can be based on performance considerations or other practical concerns. For example, the inability to differentiate ownership where a single property is used in a single operation (method) but has differing values (as far as the Licensing Service is concerned) because more than one principal is using the particular instance's method at one time.

Examples of properties that are useful:

- UNITS_TO_RESERVE provides a hint to the producer policy implementation indicating that the currently controlled aspect of the producers intellectual property has some idea about what it is going to 'use' over the next amount of time.
- VALUE_TO_CONSUMER provides a hint to the producer policy implementation indicating that the currently controlled aspect has some idea of the value of what it is currently doing.
- NODE_NAME provides a hint to the producer policy implementation about where the currently being controlled object is executing.

These are currently always producer-specific. The Licensing Service places no semantic or syntactic interpretation on these properties but makes them available, in an implementation-specific way, to the producers policy.

B.1.1 License_Use_Context

There will need to be a set of information about each producer client made available to the *ProducerSpecificLicenseService* as a "PropertySet" as specified by the Property Service. The PropertySet is a dynamic equivalent of CORBA attributes. This set of information is made available to the *start_use*, *check_use* and *end_use* operations for the Licensing System to use in determining various aspects of policy. As one example, this data structure could contain:

- All data from the *principal*, as retrieved through the new context information provided by the *CORBA 2.0* specification and as used, for example, by the Transaction Service.
- Any data the producer client may need, either in the present or the future. Being all inclusive early on reduces the need to re-deploy the licensed software if subsequent licenses need additional data.
- Fields from the example list of licensing attributes (provided below.)

The example list is useful to allow people other than the original producer to create license documents for an object implementation. This happens in the case of either acquisitions or distribution agreements. The example list makes it easier for one object implementation to be licensed by multiple license systems depending on the environment in which it finds itself.

The list items are suggestions. Currently, no central registry of names exists; also, many items are not clearly defined. The list is a starting point and can serve as a check list for Producers.

Canonical List of user_context Properties:

- DATE_TODAY
- Today's date and time.
- GROUP_ID
- Integer group
- ID GROUP_NAME
- Name of group of users
- HARDWARE_FAMILY
- String of compatible hardware family
- HARDWARE_MODEL
- Hardware model
- HARDWARE_PRODUCER
- Manufacturer name
- NETWORK_ID
- Integer network identifier
- NETWORK_NAME
- String network identifier
- NETWORK_PROTOCOL
- String protocol name, for example, "TCP/IP" or "DECnet"
- NETWORK_STYLE
- 1 is local, 2 is LAN, 3 is WAN.
- NODE_ID
- Integer node identifier
- NODE_NAME
- Name of computer
- OPERATING_SYSTEM
- String identifying the OS
- OS_VERSION
- String identifying the OS version
- PROCESS_FAMILY
- String identifying a group of related processes
- PROCESS_ID
- Integer identifying a process number
- PROCESS_NAME

- String identifying the name of the process
- PROCESS_TYPE
- 1 is batch, 2 is interactive, 3 is other
- PRODUCT_NAME
- Name of intellectual property being protected
- PRODUCT_PUBLISHER
- Owner of intellectual property being protected
- PRODUCT_VERSION
- Version string of intellectual property
- PUBLIC_KEY
- String containing public key to test against Product
- RELEASE_DATE
- Integer indicating the date the software was released
- USER_ID
- Integer indicating user
- USER_NAME
- String containing user name

B.1.2 Dependent Licenses

The Licensing Service can examine not only the most recent set of user runtime environment data but it can also examine data from previous runtime contexts collected along a particular thread of control. For example, a user may log in as "Fred" and begin some action under that name. This action may include an operation being dispatched to an object implementation logged in as "root". If this second process needs to obtain a license which was reserved for "Fred" then it ought to be able to do so. The user should be known by all the names associated with that thread of control.

Another example of a recursive license right is the "embedded" license. Such a license is not valid unless another object implementation was used earlier on the thread of control. A database software vendor might issue License Documents for use within, say, an accounting package. Other uses which might be worth more must be licensed separately. An example of an interface which would support a stack of License Use Context is as follows:

```
interface UserContext {
   Property::PropertySet  License_Use_Context  create ( );
   void push( in Property::PropertySet License_Use_Context);
   void pop ();
   unsigned long getDepth ();
   Property::PropertySet License_Use_Context   top ();
   Property::PropertySet License_Use_Context get (in unsigned long
which_frame );

   void clear ();
void remove ();
}
```

B.2 Relationship Service

Support for collections and relationships will be determined by the mechanisms made available to producers by the particular implementations of the Licensing Service. It is expected that the preferred mechanisms will be to allow the Producer Policy to make use of Object Services such as the Relationship and Property Services, but this is not a requirement of the Licensing Service.

Each implementation of the Licensing Service can address the problem of how to manage the relationships among licenses. The types of relations one can assume exists among licenses can be generically classified as follows:

- Prerequisite licenses, for example. the previous example of a database vendor.
- Corequisite licenses, that is, a set of licenses which must all coexist to give the producer client the right to be running.
- Exrequisite licenses, that is, a set of licenses that can run only if others are not active.
- Generic dependent licenses, that is, a set of licenses whose dependencies are described through a specific constraint expression.

B.3 Security Service

The Security Service will probably replace the logic in each Licensing System that deals with producer client authentication and access control.

Appendix C Producer Client Implementation Issues

C.1 Client Implementation

In this example, a Producer decides to control method activation. In the Producer's object activation, the implementation performs the obtain_producer_specific_license_service operation on the *LicenseServiceManager* interface and stores the resultant object reference. In the implementation of each method that is to be controlled, the start_use operation is performed on the stored object reference.

Depending on whether asynchronous events are supported, the Producer implementation will vary as follows:

- If true asynchronous events are supported, the Producer implementation needs to provide an interface inherited from CosEventComm, the *PushConsumer* interface.
- If asynchronous events are not supported, or the Producer chooses to not use events, then each implementation that uses the start_use operation needs to use the check_use operation no less frequently than the period specified in the recommended_check_interval argument until the implementation performs an end_use operation. If, within the recommended check interval, the Producer Client does not perform the check_use operation, the Producer Policy may choose to release the associated licenses assuming that the Producer Client has ceased functioning.

Producers must decide how they want to use the Property Service to provide properties to the start_use, check_use and end_use operations. In the Producer implementation, the returned argument action_to_be_taken from the check_use operation needs to influence how the object continues after each check_use operation.

The Producer needs to determine the name for each component and the version for each component. The Producer will then need to produce the Licensing System implementation dependant policy and license document for the Producer's chosen policy.

When a particular use of the Producer object is completed the end_use operation is used to let the Licensing Service know that control is no longer required for that component.

C.2 Asynchronous Events

In CORBA implementations where true asynchronous events are supported, provision is made in the start_use operation to provide the Licensing Service with the object reference that corresponds to a client *PushConsumer* interface. This will allow the license service to asynchronously send a push event to the specified interface with the arguments defined in the following pseudocode:

C.3 Pseudocode

```
struct AsyncLicenseData{
    ProducersSpecificLicenseService service;
    LicenseHandle handle;
    Challenge challenge;
};

/* Producer client implements an interface for the 'push' operation: */

void xxxx_push(Object o, Environment *e, any data)
{

    struct AsyncLicenseData *check;


    /* get the actual information that is needed to proceed */

    check = (struct AsyncLicenseData *)(data->_value);

    /*
    perform producer specific testing and lookup on:

        check->handle
        check->challenge

    need to make sure that the component of this instance
    that handle refers to is still active and that the
    challenge is valid.
     */
     /*
       providing all is well, cause a check_use operation for
       the handle. Have to assemble the challenge, decide which
       properties are important for this handle and so forth.
     */

    check->service->check_use(ev,
            check->handle,
            properties,
            interval,
            action,
            challenge);
```

```
    /* test the challenge returned and so forth */

}
```

When the Producer Client has the push operation invoked, activating the routine xxxx_push in the pseudocode example, the producer implementation should determine which aspect of the implementation is referenced by the handle argument and then invoke the check_use operation on the handle provided as one of the arguments to the push operation. At thar point, the implementation should determine if the object related to the handle is still active; determine if the challenge is valid; and then perform the check_use operation on the provided object reference. The results from this operation will indicate whether any action is to be taken and, if so, the implementation should proceed according to the Producer Policy.

Appendix D Challenge Mechanism

D.1 Default

For a producer to verify that a particular instance of the *LicenseServiceManager* is legitimate, a challenge mechanism is required. This requirement may either disappear or be reduced if the Security Service delivers a similar mechanism that can then be inherited by the *LicenseServiceManager.*

The mechanism proposed, by default, assumes the use of *shared secrets* in the producer implementations of their objects and the specific instance of the Licensing Service that is involved to control the producer's intellectual property.

The challenge mechanism is straightforward. When any operation is requested by a producer's instance a challenge structure is provided along with the normal parameters. This challenge structure consists of the MD5 of all the arguments to the operation, a random number, and a *forward secret value* known only to the producer. The Licensing Service instance for this producer can confirm that the client instance is legitimate by verifying that the challenge is correct. In return the instance of the license system sends back the MD5 of the same random number and a *reverse secret value* again known only to the producer. The instance invoking the operation on the Licensing Service can verify that the Licensing Service is legitimate by validating the generated MD5.[1] The challenge mechanism defined in the proposed interfaces supports more than one set of shared forward/reverse secrets. As part of the ChallengeData structure an index is provided, challenge_index, that allows the client to choose which shared secret set is to be used in the challenge. A conforming implementation of the LS needs to support at least four sets of shared secrets whose indices are 0 through 3.

This mechanism is not intended to be completely secure. Instead, it provides trust between the producer and the producer-specific instance of the Licensing Service. Eventually, the Security Service will probably replace the need for the challenge mechanism.

D.2 Alternative

As an alternative to the default challenge, a Producer can choose to define its own challenge protocol. By setting the challenge_protocol enumerated element of a challenge to 'producer_defined' the definition of what the challenge element represents becomes the responsibility of the producer and not the Licensing Service directly. This

1. MD5 is a message digest algorithm defined by R. Rivest in the Internet RFC 1321. It is in the public domain and provides a mechanism to generate a 128-bit "fingerprint" of messages of arbitrary length. It is conjectured that the difficulty of coming up with two messages that have the same digest is 2^64 operations and that generating a specific digest for a message is 2^128 operations, making it suitable for the basis of the challenge protocol described in this specification.

will depend on the implementation of the Licensing Service, since the mechanisms available to the producer to support this are defined by the way a Licensing Service is implemented.

Note

If the object producer so chooses, the same program can be licensed by more than one Licensing System. It is simply a matter of who satisfies the challenge. In fact, the challenge mechanism supports as many Licensing Service providers as an object producer chooses to pick up. They can choose sets of challenge data to deal with particular providers and use a standard set of challenge data to get the first available service provider.

It is not guaranteed to be true that all object producers will use the same challenge mechanism. However, as long as the object producer chooses to use the default challenge, this will be the case. As soon as an object producer decides to use an alternate challenge, that will be defined by the license system provider. At that point, only that implementation of the Licensing Service can satisfy the challenge and remove the multiple service provider capability. Default challenge mechanisms **must** be supported; however, if licensing system providers offer an alternative, a producer need not use the default.

Property Service 13

13.1 Overview

13.1.1 Service Description

An object supports an interface. An interface consists of operations and attributes. The interface is statically defined in OMG IDL. Two objects are of the same type if they support the same interface.

Properties are typed, named values dynamically associated with an object, outside of the type system. There are many useful cases for properties. For example:

- **Object Classification** -- A particular document may be classified as important; it must be read by the end of the day. Another document is marginally important; it must be read by the end of the month. Yet another document is not marked important. The classification of the document was invented by the user. It is not part of the document's type. However, a user may use a standard utility to find all documents marked important.
- **Object Usage Count** -- An on-line service download utility increments a counter every time an object has been downloaded by a user. The information is associated with the object but it is not part of the object's type.

The property service implements objects supporting the *PropertySet* interface or the *PropertySetDef* interface. The *PropertySet* interface supports a set of properties. A property is two tuple of: <property_name, property_value>. property_name is a string that names the property. property_value is of type *any* and carries the value assigned to the property.

The *PropertySetDef* interface is a specialization (subclass) of the *PropertySet* interface that exposes the characteristics (or metadata) of each property (e.g. readonly or read/write access). In general, this specification will use the term *PropertySet* to refer to the collection of properties and will only use the term *PropertySetDef* when explicitly referring to operations related to property metadata.

The association of properties with an object is considered an implementation detail. This property service specification allows for the creation of *PropertySets* or *PropertySetDefs* via factory interfaces, or an object may inherit the *PropertySet* or *PropertySetDef* interfaces.

Client's Model of Properties

As with CORBA attributes, clients can get and set property values. However, with properties, clients can also dynamically create and delete properties associated with an object. Clients can manipulate properties individually or in batches using a sequence of the Property data type called Properties.

In addition, when using objects that support the *PropertySetDef* interface, clients can create and manipulate properties and their characteristics, such as the property mode. The *PropertySetDef* interface also provides operations for clients to retrieve *constraint* information about a *PropertySet*, such as allowed property types.

To aid in the client's view of properties associated with an object, the client may request a list of property names (PropertyNames) or the number of properties.

Iterators are used by the property service to return lists of properties when the number of properties exceeds that which is expected by the client. Iterators contain operations that allow clients fine-grained control over the enumeration of properties.

Object's Model of Properties

Every object that wishes to provide a property service must support either the *PropertySet* or *PropertySetDef* interface. *PropertySet* is the interface that provides operations for defining, deleting, enumerating and checking for the existence of properties. The *PropertySetDef* interface is a subclass of *PropertySet* that provides operations to retrieve *PropertySet* constraints, define and modify properties with modes, and to get and set property modes.

Subclasses of *PropertySet* or *PropertySetDef* may impose restrictions on some or all of the properties they store.

Properties are intended to be the dynamic equivalent of CORBA attributes. As such, the *PropertySet* interface provides exceptions to allow implementors to support the concepts of a *readonly* property and a *fixed* property (i.e., a property that cannot be deleted). In addition, the *PropertySetDef* interface provides operations for implementors to declare their *PropertySet* constraints to clients. This mechanism is for those implementations that need the dynamics of properties, yet want the interface control of CORBA attributes.

A *PropertySet* object may support the storage of property data types itself, or there may be a "generic" *PropertySet* implementation that handles the parsing of property data types and the memory management associated with storing properties. This is considered an implementation detail.

When a *PropertySet* object receives a define_property request from a client, it must ensure there are no property_name conflicts and then retain the property information such that the object can later respond to get_property, delete_property, and is_property_defined requests from clients.

When a *PropertySet* object receives a define_property request to an existing property from a client, it must ensure that the *any* TypeCode of the property_value of the request matches the existing property's *any* TypeCode.

Use of property modes within a *PropertySet* is an implementation issue, as clients can neither access nor modify a property mode. For example, an implementation may define some initial readonly properties at create time and raise the ReadOnlyProperty exception if a client attempts to define a new property value.

13.1.2 OMG IDL Interface Summary

The property service defines interfaces to support functionality described in the previous sections. The following table gives a high-level description of the property service interfaces.

Table 13-1 Property Service Interfaces

Interface	**Purpose**
PropertySet	Supports operations for defining, deleting, enumerating and checking for the existence of properties.
PropertySetDef	Supports operations for retrieving PropertySet constraints and getting and setting property modes.
PropertiesIterator	Supports operations to allow clients fine-grained control over the enumeration of properties.
PropertyNamesIterator	Supports operations to allow clients fine-grained control over the enumeration of property names.
PropertySetFactory	Creates PropertySets.
PropertySetDefFactory	Creates PropertySetDefs.

13.1.3 Summary of Key Features

The following are key features of the Property Service:

- Provides the ability to dynamically associate named values with objects outside the static IDL-type system.
- Defines operations to create and manipulate sets of name-value pairs or name-value-mode tuples.

The names are simple OMG IDL strings. The values are OMG IDL *any*s. The use of type *any* is significant in that it allows a property service implementation to deal with any value that can be represented in the OMG IDL-type system. The modes are similar to those defined in the *Interface Repository AttributeDef* interface.

- Designed to be a basic building block, yet robust enough to be applicable for a broad set of applications.
- Provides "batch" operations to deal with sets of properties as a whole.

  The use of "batch" operations is significant in that the systems and network management (SNMP, CMIP, ...) communities have proven such a need when dealing with "attribute" manipulation in a distributed environment.
- Provides exceptions such that *PropertySet* implementors may exercise control of (or apply constraints to) the names and types of properties associated with an object, similar in nature to the control one would have with CORBA attributes.
- Allows *PropertySet* implementors to restrict modification, addition and/or deletion of properties (readonly, fixed) similar in nature to the restrictions one would have with CORBA attributes.
- Provides client access and control of constraints and property modes.
- Does not rely on any other object services.

13.2 Service Interfaces

13.2.1 CosPropertyService Module

The CosPropertyService module defines the entire property service, which consists of data types, exceptions and the following interfaces:

- *PropertySet*
- *PropertySetDef*
- *PropertySetFactory*
- *PropertySetDefFactory*
- *PropertiesIterator*
- *PropertyNamesIterator*

Data Types

The CosPropertyService module provides a number of structure and sequence data types to manipulate *PropertySet* and *PropertySetDef* information.

```
/*****************************************************/
/*  Data Types                                       */
/*****************************************************/

 typedef string PropertyName;
 struct Property {
  PropertyName property_name;
  any property_value;
 };

 enum PropertyModeType {
   normal,
   read_only,
   fixed_normal,
   fixed_readonly,
   undefined
  };

 struct PropertyDef {
  PropertyName property_name;
  any property_value;
  PropertyModeType property_mode;
 };

 struct PropertyMode {
  PropertyName property_name;
  PropertyModeType property_mode;
 };

 typedef sequence<PropertyName> PropertyNames;
 typedef sequence<Property> Properties;
 typedef sequence<PropertyDef> PropertyDefs;
 typedef sequence<PropertyMode> PropertyModes;
 typedef sequence<TypeCode> PropertyTypes;
```

Figure 13-1 Data types

A property is a two tuple of: <property_name, property_value>. property_name is a string, which names the property. property_value is of type *any* and carries the value assigned to the property. This data type is considered the base type for dealing with property data and is used throughout the *PropertySet* interface.

Clients can manipulate properties individually or in batches using a sequence of the Property data type called Properties or, when appropriate, a sequence of the PropertyName data type called PropertyNames.

A PropertyDef is a three tuple of: <property_name, property_value, property_mode_type>. property_name is a string, which names the property. property_value is of type *any* and carries the value assigned to the property. property_mode_type is an enumeration that defines the characteristics of the property. A property definition combines property characteristics (metadata) and property data information and is used in the *PropertySetDefFactory* and *PropertySetDef* interfaces. The PropertyDef data type provides clients access and control of property metadata.

Clients can manipulate property definitions individually or in batches using a sequence of the PropertyDef data type called PropertyDefs.

A PropertyMode is a two tuple of: <property_name, property_mode_type>. property_name is a string, which names the property. property_mode_type is an enumeration that defines the characteristics of the property. The PropertyMode data type is used in the *PropertySetDef* interface and provides clients access and control of property metadata.

Clients can manipulate property modes individually or in batches using a sequence of the PropertyMode data type called PropertyModes.

There are five mutually exclusive property mode types defined:

- **Normal** means there are no restrictions to the property. A client may define new values to an existing property or delete this property.
- **Readonly** means clients can only get the property information. However, a readonly property may be deleted.
- **Fixed_Normal** means the property cannot be deleted. However, clients are free to define new values to an existing property.
- **Fixed_Readonly** means the property cannot be deleted and clients can only get the property information.
- **Undefined** is used to signify PropertyNotFound when requesting a multiple get mode request. Using this on an operation that sets the mode of a property (e.g. set_mode or define_property_with_mode) will raise the UnsupportedMode exception.

Restrictions on the property_mode_type field is an implementation issue. For example, a *PropertySetDef* implementation may choose to not support a client setting a property to the fixed_readonly mode.

Exceptions

The *PropertySet* interface supports the following exceptions.

```
/*****************************************************/
/*  Exceptions                                       */
/*****************************************************/
 exception ConstraintNotSupported{};
 exception InvalidPropertyName {};
 exception ConflictingProperty {};
 exception PropertyNotFound {};
 exception UnsupportedTypeCode {};
 exception UnsupportedProperty {};
 exception UnsupportedMode {};
 exception FixedProperty {};
 exception ReadOnlyProperty {};

 enum ExceptionReason {
  invalid_property_name,
  conflicting_property,
  property_not_found,
  unsupported_type_code,
  unsupported_property,
  unsupported_mode,
  fixed_property,
  read_only_property
 };

 struct PropertyException {
  ExceptionReason reason;
  PropertyName failing_property_name;
 };

 typedef sequence<PropertyException> PropertyExceptions;

 exception MultipleExceptions {
  PropertyExceptions exceptions;
 };
```

Figure 13-2 PropertySet interface exceptions.

- ConstraintNotSupported

  Indicates that either the allowed_property_types, allowed_properties, or allowed_property_defs parameter could not be properly supported by this *PropertySet* or *PropertySetDef.*

- InvalidPropertyName

  Indicates that the supplied property_name is not valid. For example, a property_name of length 0 is invalid. Implementations may place other restrictions on property_name.

- ConflictingProperty

  Indicates that the user is trying to modify an existing property_name with an *any* TypeCode in a property_value that is different from the current.

- PropertyNotFound

  Indicates that the supplied property_name is not in the *PropertySet*.

- UnsupportedTypeCode

  Indicates that a user is trying to define a property having an *any* TypeCode that is not supported by this *PropertySet.*

- UnsupportedProperty

  Indicates that a user is trying to define a property not supported by this *PropertySet.*

- FixedProperty

  Indicates that a user is trying to delete a property that the *PropertySet* considers undeletable.

- ReadOnlyProperty

  This indicates that a user is trying to modify a property that the *PropertySet* considers to be *readonly.*

- MultipleExceptions

  This exception is used to return a sequence of exceptions when dealing with the “batch” operations of define_properties and delete_all_properties in the *PropertySet* interface, define_properties_with_modes and set_property_modes in the *PropertySetDef* interface, create_initial_propertyset in the *PropertySetFactory* interface, and create_initial_propertysetdef in the *PropertySetDefFactory* interface. Each operation defines the valid entries that may occur in the sequence.

  A PropertyException is a two tuple of: <reason, failing_property_name>. reason is an enumeration reflecting one of the exceptions defined above. failing_property_name is a string, which names the property. The sequence of property exceptions returned as MultipleExceptions is the PropertyExceptions data type.

13.2.2 PropertySet Interface

The *PropertySet* interface provides operations to define and modify properties, list and get properties, and delete properties.

The *PropertySet* interface also provides "batch" operations, such as define_properties, to deal with sets of properties as a whole. The execution of the "batch" operations is considered best effort (i.e., not an atomic set) in that not all suboperations need succeed for any suboperation to succeed.

For define_properties and delete_properties, if any suboperation fails, a MultipleExceptions exception is returned to identify which property name had which exception.

For example, a client may invoke define_properties using three property structures. The first property could be accepted (added or modified), the second could fail due to an InvalidPropertyName, and the third could fail due to a ConflictingProperty. In this case a property is either added or modified in the *PropertySet,* and a MultipleExceptions is raised with two items in the PropertyExceptions sequence.

The get_properties and delete_all_properties "batch" operations utilize a boolean flag to identify that mixed results occurred and additional processing may be required to fully analyze the exceptions.

Making "batch" operations behave in an atomic manner is considered an implementation issue that could be accomplished via specialization of this property service.

Defining and Modifying Properties

This set of operations is used to define new properties to a *PropertySet* or set new values on existing properties.

```
/* Support for defining and modifying properties */
void define_property(
        in PropertyName property_name,
        in any property_value)
    raises(InvalidPropertyName,
         ConflictingProperty,
         UnsupportedTypeCode,
         UnsupportedProperty,
         ReadOnlyProperty);

void define_properties(
        in Properties nproperties)
    raises(MultipleExceptions);
```

Figure 13-3 Operations used to define new properties or set new values

define_property

Will modify or add a property to the *PropertySet*. If the property already exists, then the property type is checked before the value is overwritten. If the property does not exist, then the property is added to the *PropertySet*.

To change the *any* TypeCode portion of the property_value of a property, a client must first delete_property, then invoke the define_property.

define_properties

Will modify or add each of the properties in Properties parameter to the *PropertySet*. For each property in the list, if the property already exists, then the property type is checked before overwriting the value. If the property does not exist, then the property is added to the *PropertySet*.

This is a batch operation that returns the MultipleExceptions exception if any define operation failed.

Table 13-2 Exceptions Raised by Define Operations

Exception Raised	Description
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
ConflictingProperty	Indicates that the property indicated created a conflict in the type or value provided.
UnsupportedTypeCode	Indicates that the *any* TypeCode of the property_value field is not supported in this *PropertySet*.
UnsupportedProperty	Indicates that the supplied property is not supported in this *PropertySet*, either due to PropertyName restrictions or specific name-value pair restrictions.
ReadOnlyProperty	Indicates that the property does not support client modification of the property_value field.
MultipleExceptions	The PropertyExceptions sequence may contain any of the exceptions listed above, multiple times and in any order.

Listing and Getting Properties

This set of operations is used to retrieve property names and values from a *PropertySet*.

```
/* Support for Getting Properties and their Names */
unsigned long get_number_of_properties();
```

Figure 13-4 Operations used to retrieve property names and values

```
void get_all_property_names(
        in unsigned long how_many,
        out PropertyNames property_names,
        out PropertyNamesIterator rest);

any  get_property_value(
        in  PropertyName property_name)
    raises(PropertyNotFound,
          InvalidPropertyName);

boolean get_properties(
        in PropertyNames property_names,
        out Properties nproperties);

void get_all_properties(
        in unsigned long how_many,
        out Properties nproperties,
        out PropertiesIterator rest);
```

Figure 13-4 Operations used to retrieve property names and values

get_number_of_properties

Returns the current number of properties associated with this *PropertySet*.

get_all_property_names

Returns all of the property names currently defined in the *PropertySet*. If the *PropertySet* contains more than how_many property names, then the remaining property names are put into the *PropertyNamesIterator*.

get_property_value

Returns the value of a property in the *PropertySet*.

get_properties

Returns the values of the properties listed in property_names.

When the boolean flag is true, the Properties parameter contains valid values for all requested property names. If false, then all properties with a value of type tk_void may have failed due to PropertyNotFound or InvalidPropertyName.

A separate invocation of get_property for each such property name is necessary to determine the specific exception or to verify that tk_void is the correct *any* TypeCode for that property name.

This approach was taken to avoid a complex, hard to program structure to carry mixed results.

get_all_properties

Returns all of the properties defined in the *PropertySet*. If more than how_many properties are found, then the remaining properties are returned in

Table 13-3 Exceptions Raised by List and Get Properties Operations

Exception Raised	Description
PropertyNotFound	Indicates that the specified property was not defined for this PropertySet.
InvalidPropertyName	Indicates the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
MultipleExceptions	The PropertyExceptions sequence may contain any of the exceptions listed above, multiple times and in any order.

Deleting Properties

This set of operations can be used to delete one or more properties from a *PropertySet*.

```
/* Support for Deleting Properties            */
void delete_property(
        in PropertyName property_name)
    raises(PropertyNotFound,
          InvalidPropertyName,
          FixedProperty);

void delete_properties(
        in PropertyNames property_names)
    raises(MultipleExceptions);

boolean delete_all_properties();
```

Figure 13-5 Operations used to delete properties

delete_property

Deletes the specified property if it exists from a *PropertySet*.

delete_properties

Deletes the properties defined in the property_names parameter. This is a batch operation that returns the MultipleExceptions exception if any delete failed.

delete_all_properties

Variation of delete_properties. Applies to all properties.

Since some properties may be defined as fixed property types, it may be that not all properties are deleted. The boolean flag is set to false to indicate that not all properties were deleted.

A client could invoke get_number_of_properties to determine how many properties remain. Then invoke get_all_property_names to extract the property names remaining. A separate invocation of delete_property for each such property name is necessary to determine the specific exception.

Note – If the property is in a *PropertySetDef*, then the set_mode operation could be invoked to attempt to change the property mode to something other than fixed before using delete_property.

This approach was taken to avoid the use of an iterator to return an indeterminate number of exceptions.

Table 13-4 Exceptions Raised by delete_properties Operations

Exception Raised	Description
PropertyNotFound	Indicates that the specified property was not defined.
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
FixedProperty	Indicates that the *PropertySet* does not support the deletion of the specified property.
MultipleExceptions	The PropertyExceptions sequence may contain any of the exceptions listed above, multiple times and in any order.

Determining If a Property Is Already Defined

The is_property_defined operation returns true if the property is defined in the *PropertySet,* and returns false otherwise.

```
boolean is_property_defined(
        in PropertyName property_name)
    raises(InvalidPropertyName);
```

Figure 13-6 is_property_defined operation

13.2.3 PropertySetDef Interface

The *PropertySetDef* interface is a specialization (subclass) of the *PropertySet* interface. The *PropertySetDef* interface provides operations to retrieve *PropertySet* constraints, define and modify properties with modes, and to get or set property modes.

It should be noted that a *PropertySetDef* is still considered a *PropertySet*. The specialization operations are simply to provide more client access and control of the characteristics (metadata) of a *PropertySet*.

The *PropertySetDef* interface also provides "batch" operations, such as define_properties_with_modes, to deal with sets of property definitions as a whole. The execution of the "batch" operations is considered best effort (i.e., not an atomic set) in that not all suboperations need to succeed for any suboperation to succeed.

For define_properties_with_modes and set_property_modes, if any suboperation fails, a MultipleExceptions exception is returned to identify which property name had which exception.

For example, a client may invoke define_properties_with_modes using four property definition structures. The first property could be accepted (added or modified), the second could fail due to an UnsupportedMode, the third could fail due to a ConflictingProperty, and the fourth could fail due to ReadOnlyProperty. In this case a property is either added or modified in the *PropertySetDef* and a MultipleExceptions exception is raised with three items in the PropertyExceptions sequence.

The get_property_modes "batch" operation utilizes a boolean flag to signal that mixed results occurred and additional processing may be required to fully analyze the exceptions.

Making "batch" operations behave in an atomic manner is considered an implementation issue that could be accomplished via specialization of this property service.

Retrieval of PropertySet Constraints

This set of operations is used to retrieve information related to constraints placed on a *PropertySet*.

```
/* Support for retrieval of PropertySet constraints*/
void get_allowed_property_types(
        out PropertyTypes property_types);

void get_allowed_properties(
        out PropertyDefs property_defs);
```

Figure 13-7 Operations used to retrieve information related to constraints

get_allowed_property_types

Indicates which types of properties are supported by this *PropertySet*. If the output sequence is empty, then there is no restrictions on the *any* TypeCode portion of the property_value field of a Property in this *PropertySet*, unless the get_allowed_properties output sequence is not empty.

For example, a *PropertySet* implementation could decide to only accept properties that had *any* TypeCodes of tk_string and tk_ushort to simplify storage processing and retrieval.

get_allowed_properties

Indicates which properties are supported by this *PropertySet*. If the output sequence is empty, then there is no restrictions on the properties that can be in this *PropertySet*, unless the get_allowed_property_types output sequence is not empty.

Defining and Modifying Properties with Modes

This set of operations is used to define new properties to a *PropertySet* or set new values on existing properties.

```
/* Support for defining and modifying properties */
void define_property_with_mode(
        in PropertyName property_name,
        in any property_value,
        in PropertyModeType property_mode)
    raises(InvalidPropertyName,
          ConflictingProperty,
          UnsupportedTypeCode,
          UnsupportedProperty,
          UnsupportedMode,
          ReadOnlyProperty);

void define_properties_with_modes(
        in PropertyDefs property_defs)
    raises(MultipleExceptions);
```

Figure 13-8 Operations used to define new properties or values

define_property_with_mode

This operation will modify or add a property to the *PropertySet*. If the property already exists, then the property type is checked before the value is overwritten. The property mode is also checked to be sure a new value may be written. If the property does not exist, then the property is added to the *PropertySet*.

To change the *any* TypeCode portion of the property_value of a property, a client must first delete_property, then invoke the define_property_with_mode.

define_properties_with_modes

This operation will modify or add each of the properties in the Properties parameter to the *PropertySet*. For each property in the list, if the property already exists, then the property type is checked before overwriting the value. The property mode is also checked to be sure a new value may be written. If the property does not exist, then the property is added to the *PropertySet*.

This is a batch operation that returns the MultipleExceptions exception if any define operation failed.

Table 13-5 Exceptions Raised by define Operations

Exception Raised	Description
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
ConflictingProperty	Indicates that the property indicated created a conflict in the type or value provided.
UnsupportedTypeCode	Indicates that the *any* TypeCode of the property_value field is not supported in this *PropertySet*.
UnsupportedProperty	Indicates that the supplied property is not supported in this *PropertySet*, either due to PropertyName restrictions or specific name-value pair restrictions.
UnsupportedMode	Indicates that the mode supplied is not supported in this *PropertySet.*
ReadOnlyProperty	Indicates that the property does not support client modification of the property_value field.
MultipleExceptions	The PropertyExceptions sequence may contain any of the exceptions listed above, multiple times and in any order.

Getting and Setting Property Modes

This set of operations is used to get and set the property mode associated with one or more properties.

```
    /* Support for Getting and Setting Property Modes */
    PropertyModeType get_property_mode(
            in  PropertyName property_name)
        raises(PropertyNotFound,
             InvalidPropertyName);

    boolean get_property_modes(
            in PropertyNames property_names,
            out PropertyModes property_modes);

    void set_property_mode(
            in PropertyName property_name,
            in PropertyModeType property_mode)
        raises(InvalidPropertyName,
             PropertyNotFound,
             UnsupportedMode);

    void set_property_modes(
            in PropertyModes property_modes)
        raises(MultipleExceptions);
};
```

Figure 13-9 Operations used to get and set property mode

get_property_mode

Returns the mode of the property in the *PropertySet*.

get_property_modes

Returns the modes of the properties listed in property_names.

When the boolean flag is true, the property_modes parameter contains valid values for all requested property names. If false, then all properties with a property_mode_type of undefined failed due to PropertyNotFound or InvalidPropertyName. A separate invocation of get_property_mode for each such property name is necessary to determine the specific exception for that property name.

This approach was taken to avoid a complex, hard to program structure to carry mixed results.

set_property_mode

Sets the mode of a property in the *PropertySet*.

Protection of the mode of a property is considered an implementation issue. For example, an implementation could raise the UnsupportedMode when a client attempts to change a fixed_normal property to normal.

set_property_modes

Sets the mode for each property in the property_modes parameter. This is a batch operation that returns the MultipleExceptions exception if any set failed.

Table 13-6 Exceptions Raised by Get and Set Mode Operations

Exception Raised	Description
PropertyNotFound	Indicates that the specified property was not defined.
InvalidPropertyName	Indicates that the property name is invalid. (A property name of length 0 is invalid; implementations may place other restrictions on property names.)
UnsupportedMode	Indicates that the mode supplied (set operations only) is not supported in this *PropertySet.*
MultipleExceptions	The PropertyExceptions sequence may contain any of the exceptions listed above, multiple times and in any order.

13.2.4 PropertiesIterator Interface

A *PropertySet* maintains a set of name-value pairs. The get_all_properties operation of the *PropertySet* interface returns a sequence of Property structures (Properties). If there are additional properties, the get_all_properties operation returns an object supporting the *PropertiesIterator* interface with the additional properties.

The *PropertiesIterator* interface allows a client to iterate through the name-value pairs using the next_one or next_n operations.

Resetting the Position in an Iterator

The reset operation resets the position in an iterator to the first property, if one exists.

```
void reset();
```

Figure 13-10 reset operation

next_one, next_n

The *next_one* operation returns true if an item exists at the current position in the iterator with an output parameter of a property. A return of false signifies no more items in the iterator.

The next_n operation returns true if an item exists at the current position in the iterator and the how_many parameter was set greater than zero. The output is a properties sequence with at most the how_many number of properties. A return of false signifies no more items in the iterator.

```
boolean next_one(out Property aproperty);
boolean next_n(
 in unsigned long how_many,
 out Properties nproperties);
```

Figure 13-11 next_one and next_n operations (properties)

Destroying the Iterator

The destroy operation destroys the iterator.

```
void    destroy();
```

Figure 13-12 destroy operation

13.2.5 PropertyNamesIterator Interface

A *PropertySet* maintains a set of name-value pairs. The get_all_property_names operation returns a sequence of names (PropertyNames). If there are additional names, the get_all_property_names operation returns an object supporting the *PropertyNamesIterator* interface with the additional names.

The *PropertyNamesIterator* interface allows a client to iterate through the names using the next_one or next_n operations.

Resetting the Position in an Iterator

The reset operation resets the position in an iterator to the first property name, if one exists.

```
void reset();
```

Figure 13-13 reset operation

next_one, next_n

The *next_one* operation returns true if an item exists at the current position in the iterator with an output parameter of a property name. A return of false signifies no more items in the iterator.

The next_n operation returns true if an item exists at the current position in the iterator and the how_many parameter was set greater than zero. The output is a PropertyNames sequence with at most the how_many number of names. A return of false signifies no more items in the iterator.

```
boolean next_one(out PropertyName property_name);
boolean next_n(
 in unsigned long how_many,
 out PropertyNames property_names);
```

Figure 13-14 next_one, next_n operations (PropertyNames)

Destroying the Iterator

The *destroy* operation destroys the iterator.

```
void    destroy();
```

Figure 13-15 destroy operation

13.2.6 PropertySetFactory Interface

The create_propertyset operation returns a new *PropertySet*. It is considered an implementation issue as to whether the *PropertySet* contains any initial properties or has constraints.

The create_constrained_propertyset operation allows a client to create a new *PropertySet* with specific constraints. The modes associated with the allowed properties is considered an implementation issue.

The create_initial_propertyset operation allows a client to create a new *PropertySet* with specific initial properties. The modes associated with the initial properties is considered an implementation issue.

```
interface PropertySetFactory
{
  PropertySet create_propertyset();
  PropertySet create_constrained_propertyset(
          in PropertyTypes allowed_property_types,
          in Properties allowed_properties)
      raises(ConstraintNotSupported);
  PropertySet create_initial_propertyset(
          in Properties initial_properties)
      raises(MultipleExceptions);
};
```

Figure 13-16 PropertySetFactory interface

Deletion of any initial properties is an implementation concern. For example, an implementation may choose to initialize the *PropertySet* with a set of fixed_readonly properties for create_propertyset or choose to initialize all allowed_properties to be fixed_normal for create_constrained_propertyset.

The relationship of a *PropertySet* to a specific object is an implementation issue.

13.2.7 PropertySetDefFactory Interface

The create_propertysetdef operation returns a new *PropertySetDef*. It is considered an implementation issue as to whether the *PropertySetDef* contains any initial properties or has constraints.

The create_constrained_propertysetdef operation allows a client to create a new *PropertySetDef* with specific constraints, including property modes.

The create_initial_propertysetdef operation allows a client to create a new *PropertySetDef* with specific initial properties, including property modes.

```
interface PropertySetDefFactory
{
  PropertySetDef create_propertysetdef();
  PropertySetDef create_constrained_propertysetdef(
          in PropertyTypes allowed_property_types,
          in PropertyDefs  allowed_property_defs)
      raises(ConstraintNotSupported);
  PropertySetDef create_initial_propertysetdef(
          in PropertyDefs  initial_property_defs)
      raises(MultipleExceptions);
};
```

Figure 13-17 PropertySet Def Factory interface

It should be noted that deletion of intial or allowed properties is tied to the property mode setting for that property. In other words, initial or allowed properties are not inherently safe from deletion.

Appendix A Property Service IDL

The CosPropertyService module defines the entire property service, consisting of data types, exceptions, and interfaces described in previous sections.

```
module CosPropertyService
{
/*****************************************************/
/*  Data Types                                       */
/*****************************************************/

 typedef string PropertyName;
 struct Property {
  PropertyName property_name;
  any property_value;
 };

 enum PropertyModeType {
   normal,
   read_only,
   fixed_normal,
   fixed_readonly,
   undefined
  };

 struct PropertyDef {
  PropertyName property_name;
  any property_value;
  PropertyModeType property_mode;
 };

 struct PropertyMode {
  PropertyName property_name;
  PropertyModeType property_mode;
 };

 typedef sequence<PropertyName> PropertyNames;
 typedef sequence<Property> Properties;
 typedef sequence<PropertyDef> PropertyDefs;
 typedef sequence<PropertyMode> PropertyModes;
 typedef sequence<TypeCode> PropertyTypes;

 interface PropertyNamesIterator;
 interface PropertiesIterator;
 interface PropertySetFactory;
 interface PropertySetDef;
 interface PropertySet;

/*****************************************************/
/*  Exceptions                                       */
/*****************************************************/
  exception ConstraintNotSupported{};
  exception InvalidPropertyName {};
  exception ConflictingProperty {};
```

```
  exception PropertyNotFound {};
  exception UnsupportedTypeCode {};
  exception UnsupportedProperty {};
  exception UnsupportedMode {};
  exception FixedProperty {};
  exception ReadOnlyProperty {};

  enum ExceptionReason {
   invalid_property_name,
   conflicting_property,
   property_not_found,
   unsupported_type_code,
   unsupported_property,
   unsupported_mode,
   fixed_property,
   read_only_property
  };

  struct PropertyException {
   ExceptionReason reason;
   PropertyName failing_property_name;
  };

  typedef sequence<PropertyException> PropertyExceptions;

  exception MultipleExceptions {
   PropertyExceptions exceptions;
  };

/*****************************************************/
/*  Interface Definitions                           */
/*****************************************************/
 interface PropertySetFactory
 {
   PropertySet create_propertyset();
   PropertySet create_constrained_propertyset(
          in PropertyTypes allowed_property_types,
          in Properties  allowed_properties)
      raises(ConstraintNotSupported);
   PropertySet create_initial_propertyset(
          in Properties  initial_properties)
      raises(MultipleExceptions);
 };

/*---------------------------------------------------*/
 interface PropertySetDefFactory
 {
   PropertySetDef create_propertysetdef();
   PropertySetDef create_constrained_propertysetdef(
          in PropertyTypes allowed_property_types,
          in PropertyDefs  allowed_property_defs)
      raises(ConstraintNotSupported);
   PropertySetDef create_initial_propertysetdef(
          in PropertyDefs  initial_property_defs)
      raises(MultipleExceptions);
```

```
};

/*--------------------------------------------------*/
interface PropertySet
{
  /* Support for defining and modifying properties */
  void define_property(
          in PropertyName property_name,
          in any property_value)
      raises(InvalidPropertyName,
            ConflictingProperty,
            UnsupportedTypeCode,
            UnsupportedProperty,
            ReadOnlyProperty);

  void define_properties(
          in Properties nproperties)
      raises(MultipleExceptions);

  /* Support for Getting Properties and their Names */
  unsigned long get_number_of_properties();

  void get_all_property_names(
          in unsigned long how_many,
          out PropertyNames property_names,
          out PropertyNamesIterator rest);

  any  get_property_value(
          in  PropertyName property_name)
      raises(PropertyNotFound,
            InvalidPropertyName);

  boolean get_properties(
          in PropertyNames property_names,
          out Properties nproperties);

  void get_all_properties(
          in unsigned long how_many,
          out Properties nproperties,
          out PropertiesIterator rest);

  /* Support for Deleting Properties            */
  void delete_property(
          in PropertyName property_name)
      raises(PropertyNotFound,
            InvalidPropertyName,
            FixedProperty);

  void delete_properties(
          in PropertyNames property_names)
      raises(MultipleExceptions);

  boolean delete_all_properties();

  /* Support for Existence Check               */
```

```
   boolean is_property_defined(
            in PropertyName property_name)
        raises(InvalidPropertyName);
 };

/*-------------------------------------------------------*/
 interface PropertySetDef:PropertySet
 {
   /* Support for retrieval of PropertySet constraints*/
   void get_allowed_property_types(
            out PropertyTypes property_types);

   void get_allowed_properties(
            out PropertyDefs property_defs);

   /* Support for defining and modifying properties */
   void define_property_with_mode(
            in PropertyName property_name,
            in any property_value,
            in PropertyModeType property_mode)
        raises(InvalidPropertyName,
              ConflictingProperty,
              UnsupportedTypeCode,
              UnsupportedProperty,
              UnsupportedMode,
              ReadOnlyProperty);

   void define_properties_with_modes(
            in PropertyDefs property_defs)
        raises(MultipleExceptions);

   /* Support for Getting and Setting Property Modes */
   PropertyModeType get_property_mode(
            in  PropertyName property_name)
        raises(PropertyNotFound,
              InvalidPropertyName);

   boolean get_property_modes(
            in PropertyNames property_names,
            out PropertyModes property_modes);

   void set_property_mode(
            in PropertyName property_name,
            in PropertyModeType property_mode)
        raises(InvalidPropertyName,
              PropertyNotFound,
              UnsupportedMode);

   void set_property_modes(
            in PropertyModes property_modes)
        raises(MultipleExceptions);
 };

/*-------------------------------------------------------*/
 interface PropertyNamesIterator
```

```
{
  void    reset();
  boolean next_one(
            out PropertyName property_name);
  boolean next_n (
            in  unsigned long how_many,
            out PropertyNames property_names);
  void    destroy();
};

/*--------------------------------------------------*/
interface PropertiesIterator
{
  void    reset();
  boolean next_one(
            out Property aproperty);
  boolean next_n(
            in unsigned long how_many,
            out Properties nproperties);
  void    destroy();
};
};
```

Time Service Specification 14

14.1 Introduction

14.1.1 Time Service Requirements

The requirements explicitly stated in the RFP ask for a service that enables the user to obtain current time together with an error estimate associated with it.

Additionally, the RFP suggests that the service also provide the following facilities:

- Ascertain the order in which “events” occurred.
- Generate time-based events based on timers and alarms.
- Compute the interval between two events.

Although the RFP mentions specification of a synchronization mechanism, the submitters deemed it inappropriate to specify a single such mechanism as discussed in Section 14.1.3, Source of Time.

14.1.2 Representation of Time

Time is represented many ways in programs. For example the *X/Open DCE Time Servic*e [1] defines three binary representations of absolute time, while the UNIX SVID defines a different representation of time. Other systems use time represented in myriads of different ways. It is not a goal of the service defined in this submission to deal with all these different representations of time or to propose a new unifying representation of time.

To satisfy the set of requirements that are addressed, we have chosen to use only the Universal Time Coordinated (UTC) representation from the *X/Open DCE Time Service*. Global clock synchronization time sources, such as the UTC signals broadcast by the WWV radio station of the National Bureau of Standards, deliver time, which is relatively easy to handle in this representation. UTC time is defined as follows.

Time units	100 nanoseconds (10^{-7} seconds)
Base time	15 October 1582 00:00:00.
Approximate range	AD 30,000

UTC time in this service specification always refers to time in Greenwich Time Zone. The corresponding binary representations of relative time is the same one as for absolute time, and hence with similar characteristics:

Time units	100 nanoseconds (10^{-7} seconds)
Approximate range	+/- 30,000 years

In order to ease implementation on existing systems, migration from them and interoperation with them, care has been taken to ensure that the representation of time used interoperates with *X/Open DCE Time Service* [1], and that the operation for getting current time is easy to implement on *X/Open DCE Time Service*, *NTP* [2] (and for that matter any other reasonable distributed time synchronization algorithm that one might come up with, e.g. ones presented in [3]) with appropriate values for inaccuracies.

14.1.3 Source of Time

The services defined in this chapter depend on the availability of an underlying Time Service that obtains and synchronizes time as required to provide a reasonable approximation of the current time to these services. The following assumptions are made about the underlying time synchronization service:

- The Time Service is able to return current time with an associated error parameter.
- Within reasonable interpretation of the terms, the Time Service is available and reliable. The time provided by the underlying service can be trusted to be within the inaccuracy window provided by the underlying system.
- The time returned by the Time Service is from a monotonically increasing series.

Additionally, if the underlying Time Service meets the criteria to be followed for secure time presented in Appendix A, Implementation Guidelines, then the Time Service object is able to provide trusted time.

No additional assumptions are made about how the underlying service obtains the time that it delivers to this service. For example it could utilize a range of techniques whether it be using a Cesium clock attached to each node or some hardware/software time synchronization method. It is assumed that the underlying service may fail occasionally. This is accounted for by providing an appropriate exception as part of the interface. The availability and accuracy of trusted time depends on what is provided by the underlying Time Service.

14.1.4 General Object Model

The general architectural pattern used is that a service object manages objects of a specific category as shown in Figure 14-1.

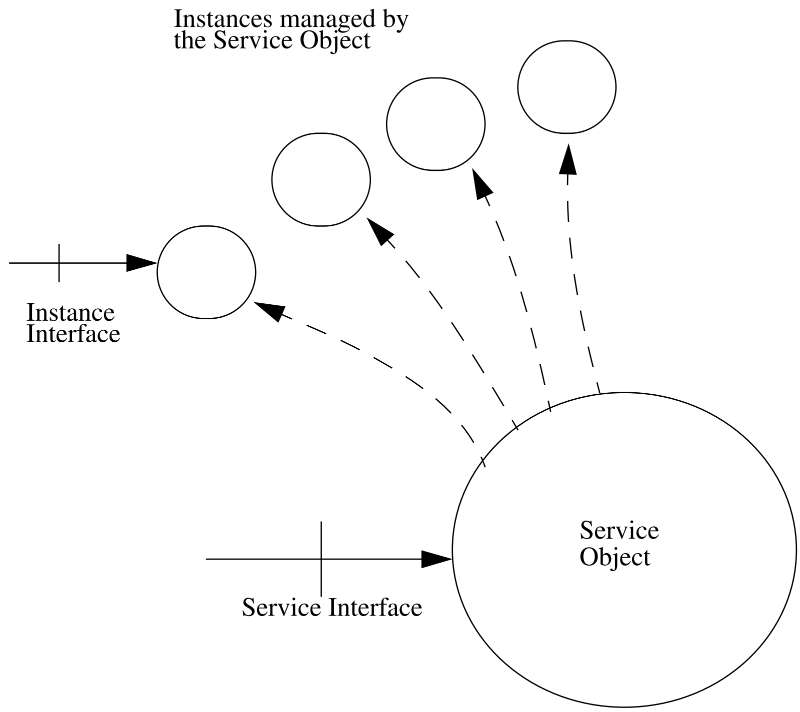


Figure 14-1 General Object Model for Service

The service interface provides operations for creating the objects that the service manages and, if appropriate, also provides operations for getting rid of them.

The Time Service object consists of two services, and hence defines two service interfaces:

- Time Service manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs), and is represented by the *TimeService* interface.
- Timer Event Service manages Timer Event Handler objects, and is represented by the *TimerEventService* interface.

The underlying facility that delivers time is associated with the **UniversalTime** and **SecureUniversalTime** operation of the *TimeService* interface as described in Section 14.2, Basic Time Service.

14.1.5 Conformance Points

There are two conformance points for this service.

- *Basic Time Service*. This service consists of all data types and interfaces defined in the TimeBase and CosTime modules in Section 14.2, Basic Time Service. It provides operations for getting time and manipulating time. A complete implementation of the TimeBase and the CosTime modules is necessary and sufficient to conform to the Time Service object standard. An implementation of the CosTime module in which the **universal_time** operation always raises the **TimeUnavailable** exception is not acceptable for satisfying this conformance point.
- *Timer Event Service*. This service consists of all data types and interfaces defined in the CosTimerEvent module in Section 14.3, Timer Event Service. It provides operations for managing time-triggered event handlers and the events that they handle. A complete implementation of this module is necessary to conform to the optional Timer Event Service component of the Time Service object. Since the CosTimerEvent module depends on the CosTime module, it is not possible to conform just to the Timer Event Service without conforming to Basic Time Service. To claim conformance to Timer Event Service, both Timer Event Service and Time Service must be provided.

14.2 Basic Time Service

All data structures pertaining to the basic Time Service, Universal Time Object, and Time Interval Object are defined in the TimeBase module so that other services can make use of these data structures without requiring the interface definitions. The interface definitions and associated enums and exceptions are encapsulated in the CosTime module.

14.2.1 Object Model

The object model of this service is depicted in Figure 14-2. The Time Service object manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs). It does so by providing methods for creating UTOs and TIOs. Each UTO represents a time, and each TIO represents a time interval, and reference to each can be freely passed around, subject to the caveats discussed in Appendix A, Implementation Guidelines.

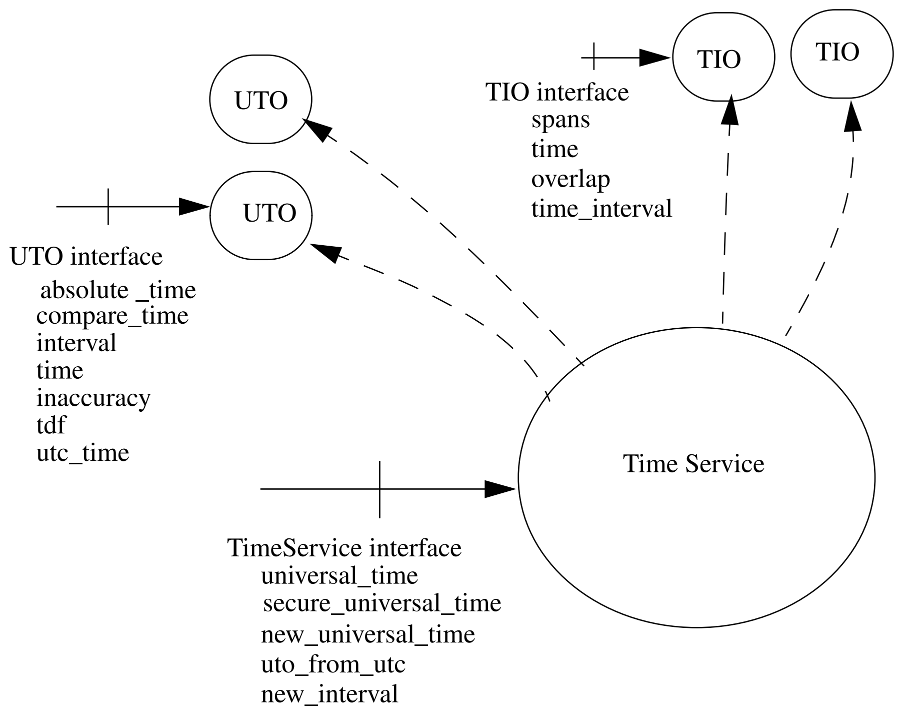


Figure 14-2 Object Model for Time Service

14.2.2 Data Types

A number of types and interfaces are defined and used by this service. All definitions of data structures are placed in the TimeBase module. All interfaces, and associated enum and exception declarations are placed in the CosTime module. This separation of basic data type definitions from interface related definitions allows other services to use the time data types without explicitly incorporating the interfaces, while allowing clients of those services to use the interfaces provided by the Time Service to manipulate the data used by those services.

```
module TimeBase {

    typedef unsigned long long          TimeT;
    typedef TimeT                       InaccuracyT;
    typedef short                       TdfT;
    struct UtcT {
        TimeT                           time;       // 8 octets
        unsigned long                   inacclo;    // 4 octets
        unsigned short                  inacchi;    // 2 octets
        TdfT                            tdf;        // 2 octets
                                                    // total 16 octets.
    };
        struct IntervalT {
            TimeT                   lower_bound;
```

```
        TimeT               upper_bound;
    };
};
```

Type TimeT

TimeT represents a single time value, which is 64 bits in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time the base is 15 October 1582 00:00 of the Gregorian Calendar. All absolute time shall be computed using dates from the Gregorian Calendar.

Type InaccuracyT

InaccuracyT represents the value of inaccuracy in time in units of 100 nanoseconds. As per the definition of the inaccuracy field in the *X/Open DCE Time Service* [1], 48 bits is sufficient to hold this value.

Type TdfT

TdfT is of size 16 bits short type and holds the time displacement factor in the form of minutes of displacement from the Greenwich Meridian. Displacements East of the meridian are positive, while those to the West are negative.

Type UtcT

UtcT defines the structure of the time value that is used universally in this service. The basic value of time is of type TimeT that is held in the time field. Whether a UtcT structure is holding a relative or absolute time is determined by its history. There is no explicit flag within the object holding that state information. The iacclo and inacchi fields together hold a 48-bit estimate of inaccuracy in the time field. These two fields together hold a value of type InaccuracyT packed into 48 bits. The tdf field holds time zone information. Implementation must place the time displacement factor for the local time zone in this field whenever they create a UTO.

The contents of this structure are intended to be opaque, but in order to be able to marshal it correctly, at least the types of fields need to be identified.

Type IntervalT

This type holds a time interval represented as two TimeT values corresponding to the lower and upper bound of the interval. An IntervalT structure containing a lower bound greater than the upper bound is invalid. For the interval to be meaningful, the time base used for the lower and upper bound must be the same, and the time base itself must not be spanned by the interval.

```
module CosTime {
    enum TimeComparison {
        TCEqualTo,
```

```
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType {
        IntervalC,
        MidC
    };

    enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
    };
};
```

Enum ComparisonType

ComparisonType defines the two types of time comparison that are supported. IntervalC comparison does the comparison taking into account the error envelope. MidC comparison just compares the base times. A MidC comparison can never return TCIndeterminate.

Enum TimeComparison

TimeComparison defines the possible values that can be returned as a result of comparing two UTOs. The values are self-explanatory. In an IntervalC comparison, TCIndeterminate value is returned if the error envelopes around the two times being compared overlap. For this purpose the error envelope is assumed to be symmetrically placed around the base time covering time-inaccuracy to time+inaccuracy. For IntervalC comparison, two UTOs are deemed to contain the same time only if the Time attribute of the two objects are equal and the Inaccuracy attributes of both the objects are zero.

Enum OverlapType

OverlapType specifies the type of overlap between two time intervals. Figure 14-3 depicts the meaning of the four values of this enum. When interval A wholly contains interval B, then it is an OTContainer of interval B and the overlap interval is the same as the interval B. When interval B wholly contains interval A, then interval A is OTContained in interval B and the overlap region is the same as interval A. When neither interval is wholly contained in the other but they overlap, then the OTOverlap case applies and the overlap region is the length of interval that overlaps. Finally, when the two intervals do not overlap, the OTNoOverlap case applies.

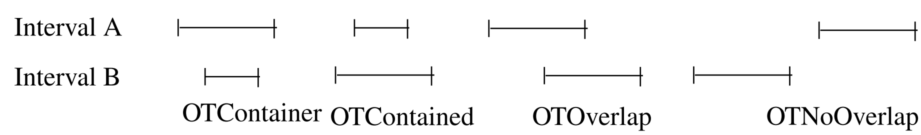


Figure 14-3 Illustration of Interval Overlap

14.2.3 Exceptions

This service returns standard CORBA exceptions where specified in addition to the service-specific exception described in this section.

```
module CosTime {
    exception TimeUnavailable {};
}
```

TimeUnavailable

This exception is raised when the underlying trusted time service fails, or is unable to provide time that meets the required security assurance.

14.2.4 Universal Time Object (UTO)

The UTO provides various operations on basic time. These include the following groups of operations:

- Construction of a UTO from piece parts, and extraction of piece parts from a UTO (as read only attributes).
- Comparison of time.
- Conversion from relative to absolute time, and conversion to an interval.

Of these, the first operation is required for completeness, since in its absence it would be difficult to provide a time input to the timer event handler, for example. The second operation is required by the RFP, and the third is required for completeness and usability.

```
module CosTime {
    interface TIO;                  // forward declaration
    interface UTO {
        readonly attribute TimeBase::TimeT              time;
        readonly attribute TimeBase::InaccuracyT        inaccuracy;
        readonly attribute TimeBase::TdfT               tdf;
        readonly attribute TimeBase::UtcT               utc_time;

        UTO absolute_time();

        TimeComparison compare_time(
```

```
            in        ComparisonType          comparison_type,
            in        UTO                     uto
        );

        TIO time_to_interval(
            in        UTO                     uto
        );

        TIO interval();
    };
};
```

The *UTO* interface corresponds to an object that contains utc time, and is the means for manipulating the time contained in the object. This interface has operations for getting a UtcT type data structure containing the current value of time in the object, as well as operations for getting the values of individual fields of utc time, getting absolute time from relative time, and comparing and doing bounds operations on UTOs. The *UTO* interface does not provide any operation for modifying the time in the object. It is intended that UTOs are immutable.

Readonly attribute time

This is the time attribute of a UTO represented as a value of type TimeT.

Readonly attribute inaccuracy

This is the inaccuracy attribute of a UTO represented as a value of type InaccuracyT.

Readonly attribute tdf

This is the time displacement factor attribute tdf of a UTO represented as a value of type TdfT.

Readonly attribute utc_time

This attribute returns a properly populated UtcT structure with data corresponding to the contents of the UTO.

Operation absolute_time

This attribute returns a UTO containing the absolute time corresponding to the relative time in object. Absolute time = current time + time in the object. Raises **CORBA::DATA_CONVERSION** exception if the attempt to obtain absolute time causes an overflow.

Operation compare_time

Compares the time contained in the object with the time given in the input parameter uto using the comparison type specified in the in parameter comparison_type, and returns the result. See the description of TimeComparison in Section 14.2.2, Data Types, for an explanation of the result. See the explanation of ComparisonType in Section 14.2.2 for an explanation of comparison types. Note that the time in the object is always used as the first parameter in the comparison. The time in the utc parameter is used as the second parameter in the comparison.

Operation time_to_interval

Returns a TIO representing the time interval between the time in the object and the time in the UTO passed in the parameter uto. The interval returned is the interval between the midpoints of the two UTOs and the inaccuracies in the UTOs are not taken into consideration. The result is meaningless if the time base used by the two UTOs are different.

Operation interval

Returns a TIO representing the error interval around the time value in the UTO as a time interval. TIO.upper_bound = UTO.time+UTO.inaccuracy. TIO.lower_bound = UTO.time - UTO.inaccuracy.

14.2.5 Time Interval Object (TIO)

The TIO represents a time interval and contains operations relevant to time intervals.

```
module CosTime {
    interface TIO {
        readonly attribute TimeBase::IntervalT time_interval;

        OverlapType spans (
            in    UTO                          time,
            out TIO                            overlap
        );
        OverlapType overlaps (
            in    TIO                          interval,
            out TIO                            overlap
        );

        UTO time ();
    }
}
```

Readonly attribute time_interval

This attribute returns an IntervalT structure with the values of its fields filled in with the corresponding values from the TIO.

Operation spans

This operation returns a value of type OverlapType depending on how the interval in the object and the time range represented by the parameter UTO overlap. See the definition of OverlapType in Section 14.2.2, Data Types. The interval in the object is interval A and the interval in the parameter UTO is interval B. If OverlapType is not OTNoOverlap, then the out parameter overlap contains the overlap interval, otherwise the out parameter contains the gap between the two intervals. The exception **CORBA::BAD_PARAM** is raised if the UTO passed in is invalid.

Operation overlaps

This operation returns a value of type OverlapType depending on how the interval in the object and interval in the parameter TIO overlap. See the definition of OverlapType in Section 14.2.2, Data Types. The interval in the object is interval A and the interval in the parameter TIO is interval B. If OverlapType is not OTNoOverlap, then the out parameter overlap contains the overlap interval, otherwise the out parameter contains the gap between the two intervals. The exception **CORBA::BAD_PARAM** is raised if the TIO passed in is invalid.

Operation time

Returns a UTO in which the inaccuracy interval is equal to the time interval in the ITO and time value is the midpoint of the interval.

14.2.6 Time Service

The *TimeService* interface provides operations for obtaining the current time, constructing a UTO with specified values for each attribute, and constructing a TIO with specified upper and lower bounds.

```
module CosTime {
    interface TimeService {
        UTO universal_time()
                raises(TimeUnavailable
        );
        UTO secure_universal_time()
                raises(TimeUnavailable
        );
        UTO new_universal_time(
                in TimeBase::TimeT              time,
                in TimeBase::InaccuracyT        inaccuracy,
                in TimeBase::TdfT               tdf
        );
        UTO uto_from_utc(
                in TimeBase::UtcT               utc
        );
```

```
        TIO new_interval(
                in TimeBase::TimeT          lower,
                in TimeBase::TimeT          upper
        );
    };
};
```

Operation universal_time

The **universal_time** operation returns the current time and an estimate of inaccuracy in a UTO. It raises TimeUnavailable exceptions to indicate failure of an underlying time provider. The time returned in the UTO by this operation is not guaranteed to be secure or trusted. If any time is available at all, that time is returned by this operation.

Operation secure_universal_time

The **secure_universal_time** operation returns the current time in a UTO only if the time can be guaranteed to have been obtained securely. In order to make such a guarantee, the underlying Time Service must meet the criteria to be followed for secure time, presented in Appendix A, Implementation Guidelines. If there is any uncertainty at all about meeting any aspect of these criteria, then this operation must return the TimeUnavailable exception. Thus, time obtained through this operation can always be trusted.

Operation new_universal_time

The **new_universal_time** operation is used for constructing a new UTO. The parameters passed in are the time of type TimeT and inaccuracy of type InaccuracyT. This is the only way to create a UTO with an arbitrary time from its components. This is expected to be used for building UTOs that can be passed as the various time arguments to the Timer Event Service, for example. **CORBA::BAD_PARAM** is raised in the case of an out-of-range parameter value for inaccuracy.

Operation uto_from_utc

The **uto_from_utc** operation is used to create a UTO given a time in the UtcT form. This has a single in parameter UTC, which contains a time together with inaccuracy and tdf. The UTO returned is initialized with the values from the UTC parameter. This operation is used to convert a UTC received over the wire into a UTO.

Operation new_interval

The **new_interval** operation is used to construct a new TIO. The parameters are lower and upper, both of type TimeT, holding the lower and upper bounds of the interval. If the value of the lower parameter is greater than the value of the upper parameter, then a **CORBA::BAD_PARAM** exception is raised.

14.3 Timer Event Service

The module CosTimerEvent encapsulates all data type and interface definitions pertaining to the Timer Event Service.

14.3.1 Object Model

The TimerEventService object manages Timer Event Handlers represented by Timer Event Handler objects as shown in Figure 14-4. Each Timer Event Handler is immutably associated with a specific event channel at the time of its creation. The Timer Event Handler can be passed around as any other object. It can be used to program the time and content of the events that will be generated on the channel associated with it. The user of a Timer Event Handler is expected to notify the Timer Event Service when it has no further use for the handler.

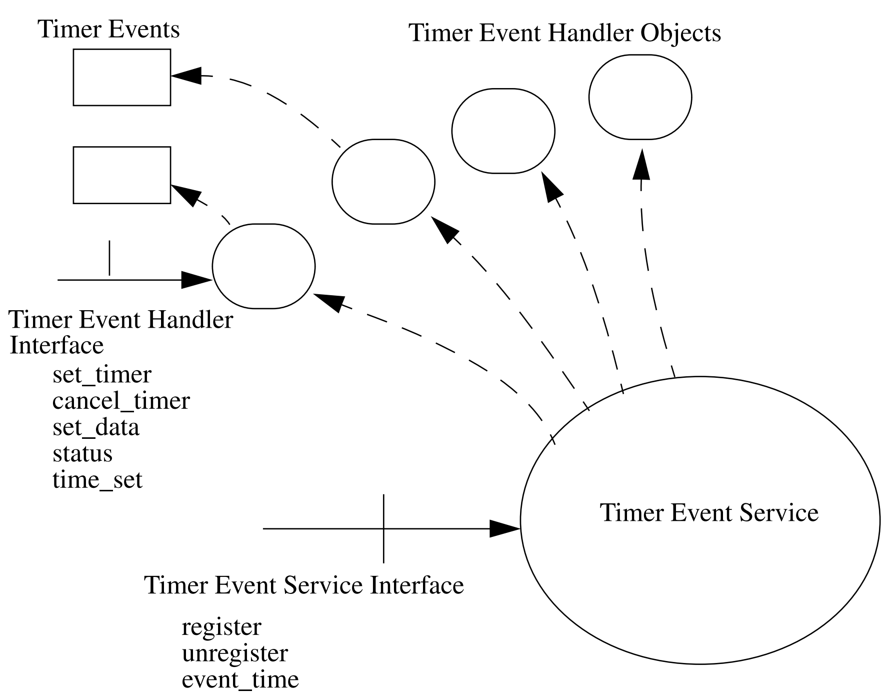


Figure 14-4 Object Model of Timer Event Service

14.3.2 Usage

In a typical usage scenario of this service, the user must first create an event channel of the "push" type (see *CORBA Service: Event Service Specification* [Chapter 4]). The user must then register this event channel as the sink for events generated by the timer event handler that is returned by the registration operation. The user can then use the timer event handler object to set up timer events as desired. The service will cause events to be pushed through the event channel within a reasonable interval around the requested event time. The implementor of the service will document what the expected interval is for their implementation. The data associated with the event includes a timestamp of the actual event time with the error envelope including the requested event time.

14.3.3 Data Types

All declarations pertaining to this service is encapsulated in the CosTimerEvent module.

```
module CosTimerEvent{
    enum TimeType {
        TTAbsolute,
        TTRelative,
        TTPeriodic
    };

    enum EventStatus {
        ESTimeSet,
        ESTimeCleared,
        ESTriggered,
        ESFailedTrigger
    };

    struct TimerEventT{
        TimeBase::UtcT          utc;
        any                     event_data;
    };
};
```

Enum TimeType

TimeType is used to specify whether a time is TTRelative, TTAbsolute, or TTPeriodic in operations for setting timer intervals for the event-triggering mechanism. The TTRelative value is used to specify that the time provided is relative to current time, TTAbsolute is used to specify that the time provided is absolute, and TTPeriodic is used to specify that the time provided is a period (and hence a relative time) between successive events. If TTPeriodic is used, then the same event continues to be triggered repeatedly at the completion of the time interval specified, until the timer is reset.

Enum EventStatus

EventStatus defines the state of a TimerEventHandler object. The state ESTimeSet means that the event has been set with a time in the future, and will be triggered when that time arrives. ESTimeCleared means that the event is not set to go off, and the time was cleared before the previously set triggering time arrived. ESTriggered means that the event has already triggered and the appropriate data has been sent the event channel. ESFailedTrigger means that the event did trigger, but data could not be delivered over the event channel.

In case of TTPeriodic events, the status ESTriggered never occurs. Upon successful triggering of a TTPeriodic event, the status is set to ESTimeSet.

Type TimerEventT

This is the structure that is returned to the event requester by the time-driven event-triggering mechanism. It has two fields. The first field, utc, contains the actual time at which the event was triggered. This value is set in the time field of utc. The inaccuracy fields inacclo and inacchi of utc are set to the difference between the requested event time and the actual event time.

The second field, event_data, contains the data that the requester of the event had asked to be sent when the event was triggered.

14.3.4 Exceptions

Timer Event Service raises standard CORBA exceptions as specified in OMG IDL for the service. It does not have any service-specific exceptions.

14.3.5 Timer Event Handler

Timer Event Handlers are created and managed by the Timer Event Service. A TimerEventHandler object holds information about an event that is to be triggered at a specific time and action that is to be taken when the event is triggered. It provides operations for setting, resetting, and canceling the timer event associated with it, as well as for changing the event data that is sent back as a part of a TimeEventT structure on the event channel upon the triggering of the event. The only thing that cannot be changed is the event channel associated with that event handler. An attribute named status holds the current status of the event handler.

```
module CosTimerEvent {
    interface TimerEventHandler {
        readonly attribute EventStatus              status;
        boolean time_set(
            out CosTime::UTO                uto
        );
        void set_timer(
            in TimeType                     time_type,
            in CosTime::UTO                 trigger_time
      );
        boolean cancel_timer();
        void set_data(
            in any                          event_data
        );
    };
};
```

Attribute status

status is a readonly attribute that reflects the current state of the TimerEventHandler. See the definition of EventStatus enumerator in Section 14.3.1, Object Model, for details.

Operation time_set

Returns **TRUE** if the time has been set for an event that is yet to be triggered, **FALSE** otherwise. In addition, it always returns the current value of the timer in the event handler as the out uto parameter.

Operation set_timer

Sets the triggering time for the event to the time specified by the uto parameter, which may contain TTRelative, TTAbsolute or TTPeriodic time. The time_type parameter specifies what type of time is contained in the uto parameter. The previous trigger, if any, is canceled and a new trigger is enabled at the time specified if absolute, or at current time + time specified if relative. If a relative time value of zero is specified (i.e. the time attribute of utc = 0LL), then the last relative time that was specified is reused. If no relative time was previously specified, then a **CORBA::BAD_PARAM** exception is raised. If a periodic time is specified (time_type == periodic), then the time parameter is interpreted as a relative time and the time trigger is set at the periodicity defined by the time (i.e. at current time + time, current time + 2 * time, etc.).

Operation cancel_timer

Cancels the trigger if one had been set and had not gone off yet. Returns **TRUE** if an event is actually canceled, **FALSE** otherwise.

Operation set_data

The data that will be passed back through the event channel in a TimerEventT structure for all future triggering of the event handler is set to event_data.

14.3.6 Timer Event Service

The Timer Event Service provides operations for registering and unregistering events.

```
module CosTimerEvent {
    interface TimerEventService {

        TimerEventHandler register(
            in CosEventComm::PushConsumer       event_interface,
            in any                                  data
        );
        void unregister(
            in TimerEventhandler                timer_event_handler
        );
```

```
            CosTime::UTO event_time(
                in TimerEventT                          timer_event
            );
        };
    };
```

Operation register

The **register** operation registers the event handler specified by the data and the event_interface parameters. When the event handler is triggered, the data is delivered using the **push** operation (of the *PushConsumer* interface in Chapter 4, Event Service Specification, Section 4.3, CosEventComm Module) specified in the event_interface parameter. Only the *Push Model* is supported for timer event delivery. Note that the event handler needs to be primed with a triggering time using the **set_time** operation of the *TimerEventHandler* interface in order for an actual event to be triggered. At initialization, the time in the handler is set to current time and its state is set to ESTimeCleared, and no event is scheduled. Raises **CORBA::NO_RESOURCE** exception if lack of resources causes it to fail to register the event handler.

Operation unregister

The **unregister** operation notifies the service that the timer_event_handler will not be used any more and all resources associated with it can be destroyed. Subsequent attempts to use that object reference will raise **CORBA::INV_OBJREF**.

Operation event_time

The **event_time** operation returns a UTO containing the time at which the event contained in the timer_event structure was triggered.

14.4 Conformance

It is sufficient to provide just the Time Service (module TimeBase and CosTime) to claim conformance with the Time Service object as described in Section 14.1.5, Conformance Points. To claim conformance with the Timer Event Service, both Time Service and Timer Event Service (module CosTimerEvent) must be provided.

In order to conform to the Basic Time Service, the semantics of the **secure_universal_time** operation must be strictly adhered to. In order to return a valid time from this operation, the vendor must provide a statement about how the security assurance criteria specified in Appendix A, Implementation Guidelines, are met in their product. To conform to the object Time Service, in all other cases, i.e. when the security assurance criteria are not satisfied, the **secure_universal_time** operation must raise the TimeUnavailable exception.

Appendix A Implementation Guidelines

A.1 Introduction

This appendix contains advice to implementors. Appropriate documented handling of the criteria presented here is mandatory for conformance to the Basic Time Service conformance point.

A.2 Criteria to Be Followed for Secure Time

The following criteria must be followed in order to assure that the time returned by the **secure_universal_time** operation is in fact secure time. If these criteria are not satisfactorily addressed in an ORB, then it must return the TimeUnavailable exception upon invocation of the **secure_universal_time** operation of the *TimeService* interface.

Administration of Time

Only administrators authorized by the system security policy may set the time and specify the source of time for time synchronization purposes.

Protection of Operations and Mandatory Audits

The following types of operations must be protected against unauthorized invocation. They must also be mandatorily audited:

- Operations that set or reset the current time
- Operations that designate a time source as authoritative
- Operations that modify the accuracy of the time service or the uncertainty interval of generated timestamps

Synchronization of Time

Synchronization of time must be transmitted over the network. This presents an opportunity for unauthorized tampering with time, which must be adequately guarded against. Time Service implementors must state how time values used for time synchronization are protected while they are in transit over the network.

Time Service implementors must state whether or not their implementation is secure. Implementors of secure time services must state how their system is secured against threats documented in Chapter 15, Security Service Specification. They must also document how the issues mentioned in this section are addressed adequately.

A.3 Proxies and Time Uncertainty

The Time Service object returns a timestamp, which contains both a time and an associated uncertainty interval. These values are considered valid at the instant they are returned by the Time Service object; however, if these values are not delivered to the caller immediately, they may no longer be reliable by the time the caller receives them.

In a CORBA system, the use of proxy objects can render time values unreliable by introducing unpredictable and uncorrected latency between the time the time server object generates a timestamp and the time the caller's time server proxy receives the timestamp and returns it to the caller (see Figure 14-5 below).

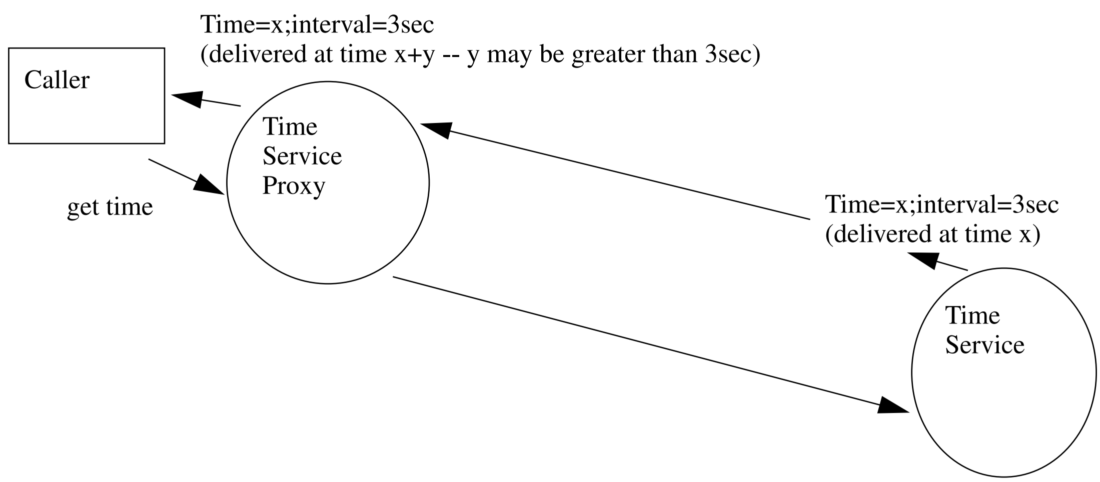


Figure 14-5 Time Service and Proxies

Implementors of the Time Service must prevent this problem from occurring. Two possible ways of preventing proxy latency are:

- Prohibit proxies of the time server object (i.e. require a Time Service implementation in every address space that will need to make Time Service calls).
- Create a special time server proxy, which measures latency between the Time Service object and the proxy, recalculates the time interval's uncertainty, and adjusts the interval value before returning the timestamp to the caller.

Other approaches probably exist; the two above are intended as examples only.

Appendix B Consolidated OMG IDL

B.1 Introduction

This appendix contains a summary of the OMG IDL defined in this document.

B.2 Time Service

This section contains the OMG IDL definitions pertaining to the Time Service, which is encapsulated in the TimeBase and CosTime modules. The TimeBase module contains the basic data type declarations that can be used by others without pulling in the Time Service interfaces. The *Time Service* interface and associated enums and exceptions are declared in the CosTime module.

```
module TimeBase {
    typedef unsigned long long          TimeT;
    typedef TimeT                       InaccuracyT;
    typedef short                       TdfT;
    struct UtcT {
        TimeT                           time;       // 8 octets
        unsigned long                   inacclo;    // 4 octets
        unsigned short                  inacchi;    // 2 octets
        TdfT                            tdf;        // 2 octets
                                                    // total 16 octets.
    };

    struct IntervalT {
        TimeT               lower_bound;
        TimeT               upper_bound;
    };
};


module CosTime {

    enum TimeComparison {
        TCEqualTo,
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType{
        IntervalC,
        MidC
    };

    enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
    };
```

```
        exception TimeUnavailable {};
        interface TIO;                  // forward declaration

        interface UTO {
            readonly attribute TimeBase::TimeT          time;
            readonly attribute TimeBase::InaccuracyTinaccuracy;
            readonly attribute TimeBase::TdfT               tdf;
            readonly attribute TimeBase::UtcT               utc_time;
            UTO absolute_time();
            TimeComparison compare_time(
                    in      ComparisonType      comparison_type,
                    in      UTO         uto
            );
            TIO time_to_interval(
                        in      UTO                     uto
            );
            TIO interval();
        };

        interface TIO {
            readonly attribute TimeBase::IntervalT time_interval;
            boolean spans (
                    in      UTO                 time,
                    out TIO                                 overlap
            );
            boolean overlaps (
                    in      TIO                 interval,
                    out TIO                     overlap
            );
            UTO time ();
        };

        interface TimeService {
            UTO universal_time()
                    raises(TimeUnavailable
            );
            UTO secure_universal_time()
                    raises(TimeUnavailable
            );
            UTO new_universal_time(
                    in TimeBase::TimeT          time,
                    in TimeBase::InaccuracyT    inaccuracy,
                    in TimeBase::TdfT           tdf
            );
            UTO uto_from_utc(
                    in TimeBase::UtcT           utc
            );
            TIO new_interval(
                    in TimeBase::TimeT          lower,
                    in TimeBase::TimeT          upper
            );
        };
    };
```

B.3 Timer Event Service

This section contains all the OMG IDL definitions pertaining to the Timer Event Service, which are encapsulated in the CosTimerEvent module. This module depends on TimeBase, CosTime, CosEventComm and CORBA.

```
module CosTimerEvent{
    enum TimeType {
                TTAbsolute,
                TTRelative,
                TTPeriodic
    };

    enum EventStatus {
                ESTimeSet,
                ESTimeCleared,
                ESTriggered,
                ESFailedTrigger
    };

    struct TimerEventT {
        TimeBase::UtcT                    utc;
        any                               event_data;
    };

    interface TimerEventHandler {
        readonly attribute EventStatus status;
        boolean time_set(
            out CosTime::UTO              uto
        );
        void SetTimer(
            in TimeType                   time_type,
            in CosTime::UTO               trigger_time
        );
        boolean cancel_timer();
        void set_data(
            in any                        event_data
        );
    };

    interface TimerEventService {
        TimerEventHandler register(
            in CosEventComm::PushConsumer event_interface,
            in any                        data
        );
        void unregister(
            in TimerEventHandler        timer_event_handler
        );
        CosTime::UTO event_time(
            in TimerEventT          timer_event
        );
    };
};
```

Appendix C Notes for Users

C.1 Introduction

This appendix contains notes covering the following matters:

- Guarding against proxy-related inaccuracies in time contained in UTO.
- How to transmit time and time intervals across the network and recover the corresponding UTO and TIO at the other end.

C.2 Proxies and Time

As explained in Appendix B, Consolidated OMG IDL, indiscriminate use of remote proxies to obtain value of current time can lead to obtaining values of time in which the inaccuracy is incorrect due to transmission delays. Consequently, care should be taken to ensure that the local Time Service is used to obtain the value of current time.

C.3 Sending Time Across the Network

When passing small objects such as UTO and TIO from one location to another, one should be aware that each time the passed object reference is used by the recipient it causes an object invocation to take place across the network and is inherently inefficient. The preferred way of dealing with this problem is to pass small objects by value instead of by reference. Unfortunately, due to various reasons, OMG IDL does not allow specification of passing of object parameters by value. Consequently, the user has to explicitly take action to avoid this problem.

The interfaces defined contain features that make it possible for the user to explicitly send the value of time, and time interval across from one location to another and then reconstruct the appropriate object at the receiving end. This is done as follows:

- The signature of the operation that passes time or time interval as a parameter across the network should specify that time is passed as the data type and not as an object reference. For example, for passing universal time, a signature such as

  ```
  void foo(in TimeBase::UtcT);
  ```

  should be used instead of

  ```
  void foo(in CosTime::UTO);
  ```

- The invoker should use the data attribute of the UTO as the in parameter. In pseudo-code, something such as the following should be done by the invoker:

  ```
  CosTime::UTO uto = CosTime::universal_time();
  foo(uto.data);
  ```

- At the server end, the time data received can be converted to a UTO as follows:

```
foo(in TimeBase::UtcT utc) {
    CosTime::UTO uto = CosTime::TimeService::uto_from_utc(utc);

    .....

};
```

It would be nice to say in the definition of the **foo** operation something such as:

```
foo(in byvalue UTO uto);
```

and have the system take care of doing essentially what is described above. However, there are difficult model- and paradigm-related issues that need resolution before such a change can be coherently proposed.

Appendix D Extension Examples

D.1 Introduction

The process of constructing the contents of a TimeBase::TimeT value can be quite tedious, involving many 64-bit multiplications and additions. The CORBA Facility for Time Representation is going to provide user-friendly ways of creating TimeT data and displaying them. However, if one is planning to use only the Time Service, it will be necessary to construct some rudimentary facility to build TimeT things. This appendix shows one way of doing this as an example of how to extend this service in useful ways.

D.2 Object Model

Following the design pattern used in the rest of this service definition, the basic extension is to define a TimeI object corresponding to the TimeT structure, and extend TimeService to provide an operation for creating such objects. The TimeI object has attributes corresponding to the user-friendly representation of time such as year, month, day, hour, minute, second, microsecond, etc.

D.3 Summary of Extensions

The additions are encapsulated in the FriendlyTime module. The changes are as follows:

- Data type declaration for components of time.
- Definition of the *TimeI* interface, consisting mostly of attributes.
- Definition of the *FriendlyTime::TimeService* interface derived from the *CosTime::TimeService* interface, for adding the operation to create TimeI objects.

D.4 Data Types

The data types are self-explanatory for the purposes of setting up this example. A complete specification should state more specific properties of each of these data types.

```
module FriendlyTime {
    typedef     unsigned short          YearT; // must be > 1581
    typedef     unsigned short          MonthT; // 1 - 12
    typedef     unsigned short          DayT; // 1 - 31
    typedef     unsigned short          HourT; // 0 - 24
    typedef     unsigned short          MinuteT; // 0 - 59
    typedef     unsigned short          SecondT; // 0 - 59
    typedef     unsigned short          MicrosecondT;
}
```

D.5 Exceptions

No exceptions are defined in this module.

D.6 Friendly Time Object

The time object provides a friendly interface to the various components usually used to represent time in normal human discourse. The set of attributes used in this example are by no means exhaustive, and is used only for illustrative purposes.

```
module FriendlyTime {
	interface TimeI {
		attribute	YearT			year;
		attribute	MonthT			month;
		attribute	DayT			day;
		attribute	HourT			hour;
		attribute	MinuteT			minute;
		attribute	SecondT			second;
		attribute	MicrosecondT		microsecond;
		attribute	TimeBase::TimeT		time;
		void reset(); // set all attributes to zero
	};
};
```

The TimeI object can be viewed as a representation conversion object. The general technique for using it is to create one using the operation **CosFriendlyTime::TimeService::time** introduced in Section D.7, Extended Time Service. This creates a TimeI object with time set to zero in it. Then the **_set** operation can be used to set the values of the various attributes. Finally, the attribute time can be used to get the corresponding TimeT value.

Conversely, one can set any TimeT value in the time attribute and then get the year, month, etc. from the appropriate attributes.

The **reset** operation facilitates reuse of time objects.

D.7 Extended Time Service

CosTime::TimeService is extended by derivation to provide an operation for creating TimeI objects.

```
module FriendlyTime {
	interface TimeService : CosTime::TimeService {
		TimeI time();
	};
};
```

D.8 Epilogue

The extension provided in this appendix makes the Time Service defined in the normative part of the document more easily usable. This leads one to wonder why this extension is not part of the main body of this submission. The reason is that there is no agreement on what the most useful representative components of time are, and the feeling that in general this should be dealt with at the Common Facilities level in general. We still felt that it would be useful to illustrate how easy it is to extend the basic service to provide this ease-of-use facility, thus this appendix.

Appendix E References

- X/Open DCE Time Service, X/Open CAE Specification C310, November 1994.
- RFC 1119 Network Time Protocol, D. Mills, September 1989.
- Probabilistic Clock Synchronization, Flaviu Cristian, Distributed Computing (1989) 3: Pg. 146-158.
- OMG IDL type Extensions RFP, Andrew Watson Ed., OMG Doc. No. 95-1-35.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 95-3-31, March 31 1995 revision, Chapter 4, Event Service Specification, Section 4.2 Pg. 4-6.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 96-10-1, October 1996 revison, Chapter 15, Security Service Specification.

Security Service Specification 15

This chapter represents the Security Service v1.2 and incorporates material that was adopted in three separate specifications related to security:

- CORBA Security Rev 1.1 (formal/97-12-22)
- Common Secure Interoperability 1.0 (orbos/96-06-20)
- CORBAsecurity/SSL Interoperability (orbos/97-02-04)

All these documents are therefore superseded by this chapter.

Associated with this specification are documents ptc/98-01-02, ptc/98-01-03, and ptc/98-01-04, which contain associated changes to the CORBA Core that have been recommended jointly by the Security RTF and the Core RTF. Also associated with this specification are the outputs of the C++ and Java language mapping RTFs that had co-terminus delivery dates with the Security 1.2 RTF.

Contents

This chapter contains the following topics.

Topic	**Page**
"Introduction to Security"	15-2
"Introduction to the Specification"	15-9
"Security Reference Model"	15-18
"Security Architecture"	15-45
"Application Developer's Interfaces"	15-88
"Administrator's Interfaces"	15-130
"Implementor's Security Interfaces"	15-156

Topic	Page
"Security Interoperability Protocols"	15-172
"Secure Inter-ORB Protocol (SECIOP)"	15-204
"The SECIOP-Hosted CSI Protocols"	15-224
"SPKM Protocol"	15-231
"GSS Kerberos Protocol"	15-234
"CSI-ECMA Protocol"	15-237
"Integrating SSL with CORBA Security"	15-272
"DCE-CIOP with Security"	15-273
Appendix A - "Consolidated OMG IDL"	15-283
Appendix B- "Relationship to Other Services"	15-312
Appendix C - "Guidelines for a Trustworthy System"	15-329
Appendix D - "Conformance Statement"	15-352
Appendix E - "Facilities Not in This Specification"	15-358
Appendix F - "Interoperability Guidelines"	15-368
Appendix G - "Glossary"	15-373
Appendix H - "References"	15-384

Note – See "Introduction to the Specification" on page 15-9 for a detailed description of the contents.

15.1 Introduction to Security

15.1.1 Why Security?

Enterprises are increasingly dependent on their information systems to support their business activities. Compromise of these systems either in terms of loss or inaccuracy of information or competitors gaining access to it can be extremely costly to the enterprise.

Security breaches, which compromise information systems, are becoming more frequent and varied. Security breaches are often due to accidental misuse of the system, such as users accidentally gaining unauthorized access to information. Commercial as well as government systems may also be subject to malicious attacks (for example, to gain access to sensitive information).

Distributed systems are more vulnerable to security breaches than the more traditional systems, as there are more places where the system can be attacked. Therefore, security is needed in CORBA systems, which takes account of their inherent distributed nature.

15.1.2 What Is Security?

Security protects an information system from unauthorized attempts to access information or interfere with its operation. It is concerned with:

- ***Confidentiality***. Information is disclosed only to users authorized to access it.
- ***Integrity***. Information is modified only by users who have the right to do so, and only in authorized ways. It is transferred only between intended users and in intended ways.
- ***Accountability***. Users are accountable for their security-relevant actions. A particular case of this is non-repudiation, where responsibility for an action cannot be denied.
- ***Availability***. Use of the system cannot be maliciously denied to authorized users.

Availability is often the responsibility of other OMA components such as archive/ restore services, or of underlying network or operating systems services. Therefore, this specification does not address all availability requirements.

Security is enforced using security functionality as described below. In addition, there are constraints on how the system is constructed, for example, to ensure adequate separation of objects so that they don't interfere with each other, and separation of users' duties so that the damage an individual user can do is limited.

Security is pervasive, affecting many components of a system, including some that are not directly security-related. Also, specialized components, such as an authentication service, provide services that are specific to security.

The assets of an enterprise need to be protected against perceived threats. The amount of protection the enterprise is prepared to pay for depends on the value of the assets, and the threats that need to be countered. The security policy needed to protect against these threats may also depend on the environment and how vulnerable the assets are in this environment. This specification specifies a security architecture that supports a variety of security policies to meet different needs.

15.1.3 Threats in a Distributed Object System

The CORBA security specification is designed to allow implementations to provide protection against the following:

- An authorized user of the system gaining access to information that should be hidden from him.
- A user masquerading as someone else, and so obtaining access to whatever that user is authorized to do, so that actions are being attributed to the wrong person. In a distributed system, a user may delegate his rights to other objects, so they can act on his behalf. This adds the threat of rights being delegated too widely, again causing a threat of unauthorized access.
- Security controls being bypassed.
- Eavesdropping on a communication line, so gaining access to confidential data.

- Tampering with communication between objects - modifying, inserting, and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

Note that some of this protection is dependent on the CORBA security implementation being constructed in the right way according to assurance criteria (as specified in Appendix D, "Guidelines for a Trustworthy System" on page 15-329) and using security mechanisms with the right characteristics. Conformance to the CORBA security interfaces is not enough to ensure that this protection is provided, just as conformance to the transactional interfaces (for example) is not enough to guarantee transactional semantics.

This specification does not attempt to counter all threats to a distributed system. For example, it does not include facilities to counter breaches caused by analyzing the traffic between machines.

More information about security threats and countermeasures is given in Appendix D, "Guidelines for a Trustworthy System" on page 15-329.

15.1.4 Summary of Key Security Features

The security functionality defined by this specification comprises:

- ***Identification*** and ***authentication*** of principals (human users and objects which need to operate under their own rights) to verify they are who they claim to be.
- ***Authorization*** and ***access control*** - deciding whether a principal can access an object, normally using the identity and/or other privilege attributes of the principal (such as role, groups, and security clearance) and the control attributes of the target object (stating which principals, or principals with which attributes) can access it.
- ***Security auditing*** to make users accountable for their security-related actions. It is normally the human user who should be accountable. Auditing mechanisms should be able to identify the user correctly, even after a chain of calls through many objects.
- ***Security of communication*** between objects, which is often over insecure lower-layer communications. This requires trust to be established between the client and target, which may require authentication of clients to targets and authentication of targets to clients. It also requires integrity protection and (optionally) confidentiality protection of messages in transit between objects.
- ***Non-repudiation*** provides irrefutable evidence of actions such as proof of origin of data to the recipient, or proof of receipt of data to the sender to protect against subsequent attempts to falsely deny the receiving or sending of the data.
- ***Administration*** of security information (for example, security policy) is also needed.

This visible security functionality uses other security functionality such as *cryptography*, which is used in support of many of the other functions but is not visible outside the Security services. No direct use of cryptography by application objects is proposed in this specification, nor are any cryptographic interfaces defined.

15.1.5 Goals

The security architecture and facilities described in this specification were designed with the following goals in mind. Not all implementations conforming to this specification will meet all these goals.

15.1.5.1 Simplicity

The model should be simple to understand and administer. This means it should have few concepts and few objects.

15.1.5.2 Consistency

It should be possible to provide consistent security across the distributed object system and associated legacy systems. This includes:

- Support of consistent policies for determining who should be able to access what sort of information within a security domain that includes heterogeneous systems.
- Fitting with existing permission mechanisms.
- Fitting with existing environments. For example, the ability to provide end-to-end security even when using communication services, which are inherently insecure.
- Fitting with existing logons (so extra logons are not needed) and with existing user databases (to reduce the user administration burden).

15.1.5.3 Scalability

It should be possible to provide security for a range of systems from small, local systems to large intra- and inter-enterprise ones. For larger systems, it should be possible to:

- Base access controls on the privilege attributes of users such as roles or groups (rather than individual identities) to reduce administrative costs.
- Have a number of security domains, which enforce different security policy details but support interworking between them subject to policy. (This specification includes architecture, but not interfaces for such interdomain working.)
- Manage the distribution of cryptographic keys across large networks securely and without undue administrative overheads.

15.1.5.4 Usability for End Users

Security should be available as transparently as possible, based on sensible, configurable defaults.

Users should need to log on to the distributed system only once to access object systems and other IT services.

15.1.5.5 Usability for Administrators

The model should be simple to understand and administer and should provide a single system image. It should not be necessary for an administrator to specify controls for individual objects or individual users of an object (except where security policy demands this).

The system should provide good flexibility and fine granularity.

15.1.5.6 Usability for Implementors

Application developers should not have to be aware of security for their applications to be protected. However, a developer who understands security should be able to protect application-specific actions.

15.1.5.7 Flexibility of Security Policy

The security policy required varies from enterprise to enterprise, so choices of security features should be allowed. An enterprise should need to pay only for the level of protection it requires, reducing the level (and therefore costs) for less sensitive information or when the system is less vulnerable to threats. The enterprise should be able to balance the costs of providing security, including the resources required to implement, administer and run the system, against the perceived potential losses incurred as the result of security breaches.

Particular types of flexibility required include:

- ***Choice of access control policy***. The interfaces defined here allow for a choice of mechanisms, ACLs using a range of privilege attributes such as identities, roles, groups, or labels. Details are hidden except from some administrative functions and security-aware applications that want to choose their own mechanisms.
- ***Choice of audit policy***. The event types to be audited are configurable. This makes it possible to control the size of the audit trail, and therefore the resources required to store and manage it.
- Support for ***security functionality profiles*** as defined either in national or international government criteria such as TCSEC (the US Trusted Computer Evaluation Security Criteria) and ITSEC (the European Information Technology Security Evaluation Criteria), or by more commercial groups such as X/Open, is required.

15.1.5.8 Independence of Security Technology

The CORBA security model should be security technology neutral. For example, interfaces specified for security of client-target object invocations should hide the security mechanisms used from both the application objects and ORB (except for some security administrative functions). It should be possible to use either symmetric or asymmetric key technology.

It should be possible to implement CORBA security on a wide variety of existing systems, reusing the security mechanisms and protocols native to those systems. For example, the system should not require introduction of new cryptosystems, access control repositories, or user registries. If the system is installed in an environment that also includes a procedural security regime, the composite system should not require dual administration of the user or authorization policy information.

15.1.5.9 Application Portability

An application object should not need to be aware of security, so it can be ported to environments that enforce different security policies and use different security mechanisms. If an object enforces security itself, interfaces to Security services should hide the particular security mechanisms used (e.g., for authentication). The application security policy (for example, to control access to its own functions and state) should be consistent with the system security policy. For example, use should be made of the same attributes for access control. Portability of applications enforcing their own security depends on such attributes being available.

15.1.5.10 Interoperability

The security architecture should allow interoperability between objects to include:

- Providing consistent security across a heterogeneous system where different vendors may supply different ORBs.
- Interoperating between secure systems and those without security.
- Interoperating between domains of a distributed system where different domains may support different security policies, for example, different access control attributes.
- Interoperating across systems that support different security technology.

This specification includes an architecture that covers all of these, at least in outline; however, it does not give specific interfaces and protocols for the last two. Interoperability between domains is expected to have limited functionality in initial implementations, and interoperability between security mechanisms is not expected to be supported.

15.1.5.11 Performance

Security should not impose an unacceptable performance overhead, particularly for normal commercial levels of security, although a greater performance overhead may occur as higher levels of security are implemented.

15.1.5.12 Object Orientation

The specification should be object-oriented:

- The security interfaces should be purely object-oriented.

- The model should use encapsulation to promote system integrity and to hide the complexity of security mechanisms under simple interfaces.
- The model should allow polymorphic implementations of its objects based on different underlying mechanisms.

15.1.5.13 Specific Security Goals

In addition to the security requirements listed above, there are more specific requirements that need to be met in some systems; therefore, the architecture must take into account:

- ***Regulatory requirements***. The security model must conform to national government regulations on the use of security mechanisms (cryptography, for example). There are several types of controls, for example, controls on what can be exported and controls on deployment and use such as limitations on encryption for confidentiality. Details vary from country to country; examples of requirements to satisfy a number of these are:
  - Allowing use of different cryptographic algorithms.
  - Keeping the amount of information encrypted for confidentiality to a minimum.
  - Using identities for auditing which are anonymous, except to the auditor.
- ***Evaluation criteria for assurance***. The security functionality and architecture must allow implementations to conform to standard security evaluation criteria such as TCSEC, ITSEC, or Common Criteria (CC)[1]for security functionality and assurance (which gives the required level of confidence in the correctness and effectiveness of the security functionality). It should allow assurance and security functionality classes or profiles up to about the E3/B2 level. However, the specification also allows systems with lower levels of security, where other requirements such as performance are more important.

15.1.5.14 Security Architecture Goals

The security architecture should confine key security functionality to a trusted core, which enforces the essential part of the security policy such as:

- Ensuring that object invocations are protected as required by the security policy.
- Requiring access control and auditing to be performed on object invocation.
- Preventing (groups of) application objects from interfering with each other or gaining unauthorized access to each other's state.

1. Version 1 or 2.

It must be possible to implement this trusted computing base so it cannot be bypassed, and kept small to reduce the amount of code which needs to be trusted and evaluated in more secure systems. This trusted core is distributed, so it must be possible for different domains to have different levels of trust.

It should also be possible to construct systems where particular Security services can be replaced by ones using different security mechanisms, or supporting different security policies without changing the application objects or ORB when using them (unless these objects have chosen to do this in a mechanism or policy-specific way).

The security architecture should be compatible with standard distributed security frameworks such as those of POSIX and X/Open.

15.2 Introduction to the Specification

This specification specifies how to provide security in stand-alone and distributed CORBA-compliant systems. Introducing Object Security services does not in itself provide security in an object environment; security is pervasive, so introducing it has implications on the Object Request Broker and on most Object services, Common Facilities, and object implementations.

This specification defines the core security facilities and interfaces required to ensure a reasonable level of security of a CORBA-compliant system as a whole. The specification includes:

- A security model and architecture that describe the security concepts and framework, the security objects needed to implement them, and how this counters security threats.
- The security facilities available to applications. This includes security provided automatically by the system, protecting all applications, even those unaware of security. The security facilities can also be used by security-aware applications through OMG IDL interfaces defined in this specification.
- The security facilities and interfaces available for performing essential security administration.
- The security facilities and interfaces available to ORB implementors, to be used in the production of secure ORBs.
- A description of how Security services affect the CORBA 2 ORB interoperability protocols.
- A description of different levels of secure interoperability that are possible.
- A description of how these levels of interoperability can be provided using a select set of popular security mechanisms and protocols.

Items not included in this specification are:

- Support for interoperability between ORBs using different security mechanisms, though interoperability of different ORBs using the same security mechanism is supported.

- Audit analysis tools, though an audit service that both the system and applications can use to record events is included.
- Management interfaces other than essential security policy management interfaces, as management services are beyond the scope of this specification. The security policy management interfaces were viewed as a necessary feature of this specification, as it is not possible to deploy a secure system without defining and managing its policy.
- Interfaces to allow applications to access cryptographic functions for use, for example, in protecting their stored data. These interfaces are not provided for two reasons: first, cryptography is generally a low-level primitive, used by Security Service implementors but not needed by the majority of application developers; and second, providing a cryptographic interface would require addressing a variety of difficult regulatory and import/export issues.
- Specific security policy profiles.

The security model and architecture specified is extensible, to allow addition of further security facilities later.

15.2.1 Specification Structure

15.2.1.1 Structure of the Chapter

The structure of the chapter is summarized below in Figure 15-1.

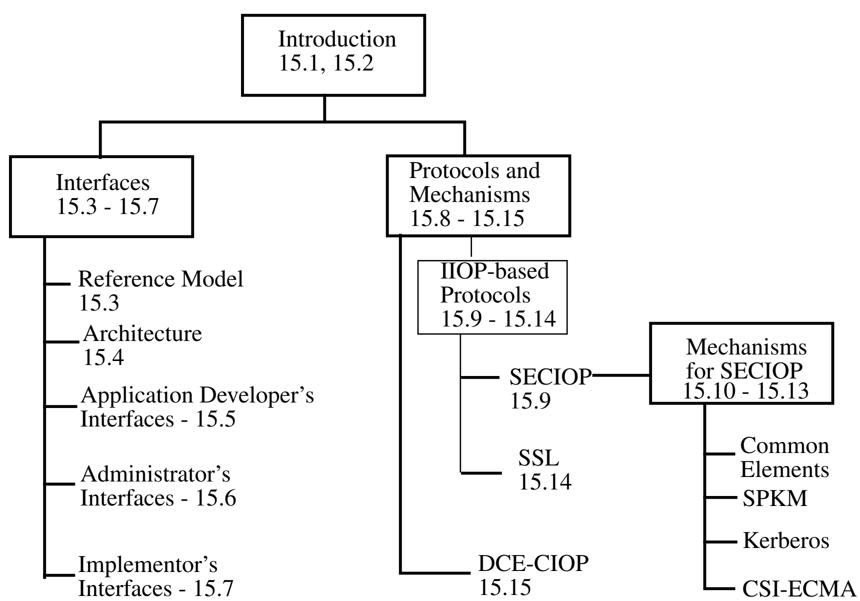


Figure 15-1 Structure of the Specification

15.2.1.2 Normative and Non-normative Material

This specification contains normative and non-normative (explanatory) material. Only 15.5 through 15.15 and Appendices A, C, and E are normative.

15.2.1.3 Section Summaries

Section 15.1 and its subsections, which is an introduction to security, explains why security is needed in distributed object systems, and enumerates the security requirements for secure distributed object systems.

Section 15.2 and its subsections provide an introduction to and overview of the specification.

Section 15.3 and its subsections describe the security reference model, which provides the overall framework for CORBA security.

Section 15.4 and its subsections describe the security architecture, which underlies this specification. This introduces different users' views of security and gives an outline of how secure CORBA-compliant systems are constructed. It also presents high-level models of the objects involved for different views, and describes how they are used.

Section 15.5 and its subsections specify the security facilities and interfaces available to **application developers**. Most functions can be implemented transparently to applications, though interfaces and additional functionality are available to security-aware applications.

Section 15.6 and its subsections specify the **administrator's** facilities and interfaces. Only essential administration functions are defined by this specification; other administrative capabilities are expected to be developed outside the Object Services Program.

Section 15.7 and its subsections specify the **implementors interfaces** used to build secure CORBA systems. This section specifies the IDL interfaces of the security objects available to ORB implementors, and describes the relationship and dependencies of these objects on the ORB core and also on external Security services, where these are used.

Section 15.8 and its subsections specify the architecture for **interoperability** in a secure, distributed object system. Further subsections lay the basic foundations for the discussion of **common secure interoperability mechanisms** in the subsequent sections. It also describes how the **common secure interoperability mechanisms** relate to the **security facilities and interfaces** presented in section 15.3 through 15.7.

Section 15.9 specifies how security is layered onto the **GIOP/IIOP** in the form of the **SECIOP** protocol.

Section 15.10 and its subsections introduce the **common elements** in the secure interoperability protocol mechanisms and how the **common elements** map to the **SECIOP protocol**.

Section 15.11 and its subsections describe how the **SPKM protocol** is used in conjunction with the **SECIOP protocol**.

Section 15.12 and its subsections describe how the **Kerberos V5 protocol** is used in conjunction with the **SECIOP protocol**.

Section 15.13 and its subsections describe the **CSI-ECMA protocol** and how it is used in conjunction with the **SECIOP protocol**.

Section 15.14 and its subsections specify how **SSL** is used as a secure transport mechanism with IIOP.

Section 15.15 and its subsections specify how security is incorporated into the **DCE-CIOP** using its Kerberos mechanism.

Appendix A, **Consolidated OMG IDL**, contains the complete OMG IDL specification, including the module structure, of the interfaces defined in this specification.

Appendix B, **Relationship to Other Services**, describes the relationship of the Security services to other object services and to the common facilities.

Appendix C, **Conformance Details**, describes in more detail what conformance to the security functionality conformance levels and the security implementation conformance points require.

Appendix D, **Guidelines for a Trustworthy System**, provides guidelines for implementation of a trustworthy system, which provides protection against the security threats in a distributed object system with the required assurance of its correctness and effectiveness.

Appendix E, **Conformance Statement**, describes the conformance statement, which must accompany a secure CORBA implementation and what this implementation must contain.

Appendix F, **Facilities Not in This Specification**, outlines security facilities that have not been included in this specification, but left for another phase of security specifications.

Appendix G, **Interoperability Guidelines**, includes guidelines for defining security mechanism tags in interoperable object references, and examples of the use of the secure inter-ORB protocol SECIOP.

Appendix H, Glossary.

Appendix I, References.

15.2.2 CORBA Security and Secure Interoperability Feature Packages

CORBA security and Secure Interoperability is structured into several feature packages which are enumerated below. These are used to structure the specification as well as to specify the conformance requirements.

15.2.2.1 Main Security Functionality Packages

There are two packages:

- *Level 1*: Provides a first level of security for applications which are unaware of security and for those having limited requirements to enforce their own security in terms of access controls and auditing.
- *Level 2*: Provides more security facilities, and allows applications to control the security provided at object invocation. It also includes administration of security policy, allowing applications administering policy to be portable.

An ORB must provide at least one of these packages before it can claim to be a Secure ORB. For a definitive conformance requirement see Appendix C, "Conformance Details" on page 15-315.

15.2.2.2 Optional Security Functionality Packages

These provide functions that are expected to be required in several ORBs, so are worth including in this specification, but are not generally required enough to form part of one of the main security functionality packages specified above.

The following is the only such option in this specification.

Non-repudiation

This provides generation and checking of evidence so that actions cannot be repudiated.

15.2.2.3 Security Replaceability Packages

These packages specify if the ORB is structured in a way that allows incorporation of different Security services, and, if so, how they can be incorporated. There are two possibilities:

ORB Services replaceability package

The ORB uses interceptor interfaces to call on object services, including security ones. It must use the specified interceptor interfaces and call the interceptors in the specified order. An ORB conforming to this does not include any significant security-specific code, as that is in the interceptors.

Security Service replaceability package

The ORB may or may not use interceptors, but all calls on Security services are made via the replaceability interfaces specified in Section 15.7, "Implementor's Security Interfaces," on page 15-156. These interfaces are positioned so that the Security services do not have to understand how the ORB works (for example, how the required policy objects are located), so they can be replaced independently of that knowledge.

An ORB can provide Security by directly implementing the Security feature package 1 or 2 into it without making use of any of the facilities provided by the Replaceability feature packages. But in that case, the standard security policies defined in this specification cannot be replaced by others, nor can the implementation of the Security services be replaced. For example, it would not be possible to replace the standard access policy with a label-based policy if at least one of the replaceability packages is not supported. Note that some replaceability of the security mechanism used for security associations may still be provided if the implementation uses some standard generic interface for Security services such as GSS-API[11].

An ORB that supports one or both of these replaceability packages together with a couple of basic ORB operations as discussed in Appendix C, "Conformance Details" on page 15-315 is said to be *Security-Ready*[2]. Such an ORB does not in itself support any security functionality but is to host security functionality that is implemented to use the facilities of the Security Replaceability package to hook Security into it.

15.2.2.4 Common Secure Interoperability (CSI) Feature packages

These feature packages each provide different levels of secure interoperability. There are three functionality levels for Common Secure Interoperability (CSI). All levels can be used in distributed secure CORBA-compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

An ORB conforming to CSI level 2 can support all the security functionality described in the CORBA Security specification. Facilities are more restricted at levels 0 and 1. The three levels are listed below.

1. Identity-based policies without delegation (CSI level 0)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

2. Identity-based policies with unrestricted delegation (CSI level 1)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in Section 15.3.6, "Delegation," on page 15-30.)

---

2. While this may sound strange, it is still true that a Secure ORB need not be a Security-Ready ORB.

3. Identity- & privilege-based policies with controlled delegation (CSI level 2):

At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

An ORB that interoperates securely must provide at least one of the CSI packages. For the definitive statement on conformance requirements see Appendix Section C.2, "Conformance Requirements," on page 15-318.

15.2.2.5 SECIOP Interoperability package

An ORB with the SECIOP Interoperability package can generate and use security information in the IOR and can send and receive secure requests to and from other ORBs using the GIOP/IIOP protocol with the security (SECIOP) enhancements defined in Section 15.9, "Secure Inter-ORB Protocol (SECIOP)," on page 15-204 (if necessary), if they both use the same underlying security technology.

15.2.2.6 Security Mechanism packages

The choice of mechanisms and protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. This specification defines how the following four security protocols can be used as the medium for secure interoperability under CORBA.

1. SPKM Protocol

Supports identity-based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20]. The use of SPKM in CORBA interoperability is based on the SECIOP extensions to IIOP.

2. GSS Kerberos Protocol

Supports identity-based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0). The GSS Kerberos protocol is based on [12] which itself is a profile of [13]. The use of Kerberos in CORBA interoperability is based on the SECIOP extensions to IIOP.

3. CSI-ECMA Protocol

Supports identity- and privilege-based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0). For keys assigned to principals, it has two options:

- It can use either secret or public key technology.

- It uses public key technology for keys assigned to trusted authorities.

  The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235. The use of CSI-ECMA in CORBA interoperability requires the SECIOP extensions to IIOP.

4. SSL protocol

Supports identity-based policies without delegation (CSI level 0). The SSL protocol is based on the definition in [21]. The use of SSL in CORBA interoperability does not depend on the SECIOP extensions to IIOP.

15.2.2.7 SECIOP Plus DCE-CIOP Interoperability

An ORB with the Standard plus DCE-CIOP secure interoperability package supports all functionality required by the standard secure interoperability package, and provides secure interoperability (using the DCE Security services) using the DCE-CIOP protocol.

An ORB that interoperates securely must do so using one of these protocol packages. For the definitive statement on conformance requirements see Appendix C, “Conformance Details” on page 15-315.

The requirements that must be satisfied by a conformant ORB are enumerated in Appendix C, “Conformance Details” on page 15-315. The conformance statement required for a CORBA-conformant security implementation is defined in Appendix E, “Conformance Statement” on page 15-352. This includes a table which can be filled to show what the ORB conforms to.

15.2.3 Feature Packages and Modules

The OMG IDL specified in this chapter is partitioned into modules that closely reflect the feature packaging scheme described above. The Security module holds definitions of common data structures and constants that most other modules depend on. The relationship is shown in Table 15-1.

Table 15-1 Feature Packages and Modules

Feature Package	Primary Module	Also Depends on
Security Functionality Level 1	SecurityLevel1	Security CORBA, TimeBase
Security Functionality Level 2	SecurityLevel2	Security, CORBA, TimeBase SecurityLevel1 SecurityAdmin
Non-Repudiation	NRservice	Security, SecurityLevel2 CORBA, TimeBase
Security Service Replaceability	SecurityReplaceable	Security, CORBA, TimeBase SecurityLevel2
ORB Service Replaceability	Interceptor	CORBA
CSI Level 0, 1, and 2	SECIOP	CORBA
SECIOP	SECIOP	Security, CORBA, TimeBase, IOP
SPKM, Kerberos, CSI-ECMA	SECIOP	Security, CORBA, TimeBase, IOP
SSL	SSL	Security, CORBA, TimeBase, IOP
DCE-CIOP	DCE_CIOPSecurity	Security, CORBA, TimeBase, IOP

The specification is based on a general three-layer architecture as shown in Figure 15-2 on page 15-18, with the interfaces defined in each module positioned as shown in the figure.

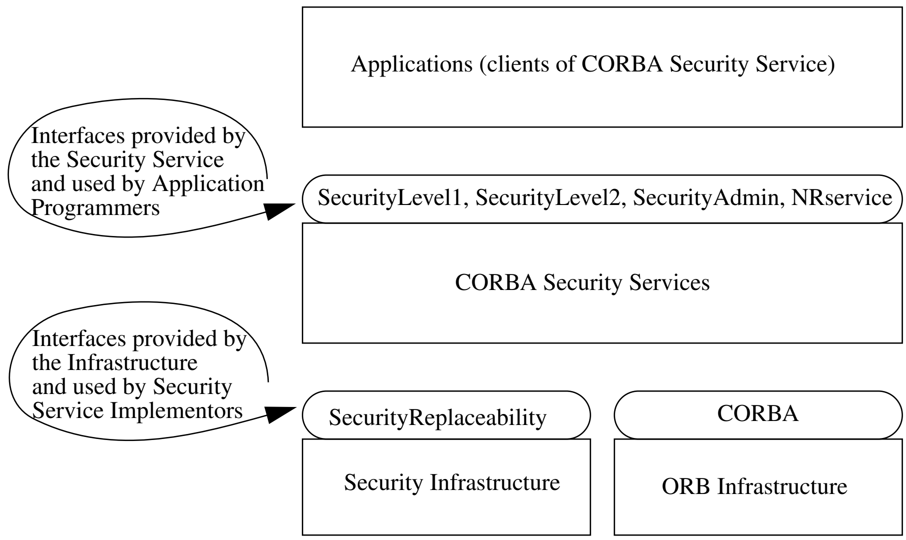


Figure 15-2 Modules and Their Relation to Layers of the Architecture

The SecurityReplaceability module defines the interfaces that must be used, together with certain interfaces defined in the SecurityLevel2 module, to encapsulate the underlying security infrastructure so as to enable components of the Security Service to use them interchangeably.

15.3 Security Reference Model

This section describes a security reference model that provides the overall framework for CORBA security. The purpose of the reference model is to show the flexibility for defining many different security policies that can be used to achieve the appropriate level of functionality and assurance. As such, the security reference model functions as a guide to the security architecture.

15.3.1 Definition of a Security Reference Model

A reference model describes how and where a secure system enforces security policies. Security policies define:

- Under what conditions active entities (such as clients acting on behalf of users) may access objects.
- What authentication of users and other principals is required to prove who they are, what they can do, and whether they can delegate their rights. (A principal is a human user or system entity that is registered in and is authentic to the system.)
- The security of communications between objects, including the trust required between them and the quality of protection of the data in transit between them.

- What accountability of which security-relevant activities is needed.

Figure 15-3 depicts the model for CORBA-secure object systems. All object invocations are mediated by appropriate security functions to enforce policies such as access controls. These functions should be tamper-proof, always be invoked when required by security policy, and function correctly.

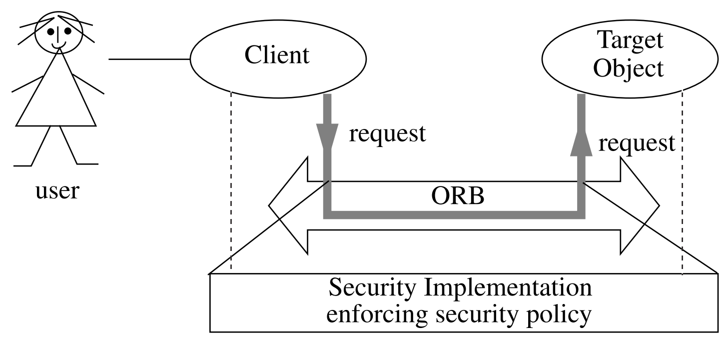


Figure 15-3 A Security Model for Object Systems

Many application objects are unaware of the security policy and how it is enforced. The user can be authenticated prior to calling the application client and then security is subsequently enforced automatically during object invocations. Some applications will need to control or influence what policy is enforced by the system on their behalf, but will not do the enforcement themselves. Some applications will need to enforce their own security, for example, to control access to their own data or audit their own security-relevant activities.

The ORB cannot be completely unaware of security as this would result in insecure systems. The ORB is assumed to at least handle requests correctly without violating security policy, and to call Security Services as required by security policy.

A security model normally defines a *specific* set of security policies. Because the OMG Object Management Architecture (OMA) must support a wide variety of different security policies to meet the needs of many commercial markets, a single instance of a security model is not appropriate for the OMA. Instead, a security reference model is defined that provides a framework for supporting many different kinds of policies. The security reference model is a *meta-policy* because it is intended to encompass all possible security policies supported by the OMA.

The meta-policy defines the abstract interfaces that are provided by the security architecture defined in this specification. The model enumerates the security functions that are defined as well as the information available. In this manner, the meta-policy provides guidance on the permitted flexibility of the policy definition. The remaining sections describe the elements of the meta-model. The description is kept deliberately general at this point.

15.3.2 Principals and Their Security Attributes

An active entity must establish its rights to access objects in the system. It must either be a principal, or a client acting on behalf of a principal.

A principal is a human user or system entity that is registered in and authentic to the system. Initiating principals are the ones that initiate activities. An initiating principal may be authenticated in a number of ways, the most common of which for human users is a password. For systems entities, the authentication information such as its long-term key, needs to be associated with the object.

An initiating principal has at least one, and possibly several identities (represented in the system by attributes) which may be used as a means of:

- Making the principal accountable for its actions.
- Obtaining access to protected objects (though other privilege attributes of a principal may also be required for access control).
- Identifying the originator of a message.
- Identifying who to charge for use of the system.

There may be several forms of identity used for different purposes. For example, the audit identity may need to be anonymous to all but the audit administrator, but the access identity may need to be understood so that it can be specified as an entry in an access control list. The same value of the identity can be used for several of the above.

The principal may also have privilege attributes which can be used to decide what it can access. A variety of privilege attributes may be available depending on access policies (see “Access Policies” on page 15-26). The privilege attributes, which a principal is permitted to take, are known by the system. At any one time, the principal may be using only a subset of these permitted attributes, either chosen by the principal (or an application running on its behalf), or by using a default set specified for the principal. There may be limits on the duration for which these privilege attributes are valid and may be controls on where and when they can be used.

Security attributes may be acquired in three ways:

1. Some attributes may be available, without authentication, to any principal. This specification defines one such attribute, called *Public*.
2. Some attributes are acquired through authentication; identity attributes and privilege attributes are in this category.
3. Some attributes are acquired through delegation from other principals.

When a user or other principal is authenticated, it normally supplies:

- Its security name.
- The authentication information needed by the particular authentication method used.
- Requested privilege attributes (though the principal may change these later).

A principal's security attributes are maintained in secure CORBA systems in a credential as shown in Figure 15-4.

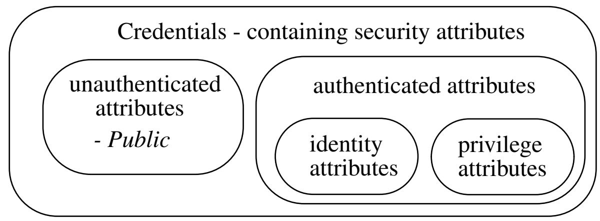


Figure 15-4 Credential Containing Security Attributes

15.3.3 Secure Object Invocations

Most actions in the system are initiated by principals (or system entities acting on their behalf). For example, after the user logs onto the system, the client invokes a target object via an ORB as shown in Figure 15-5.

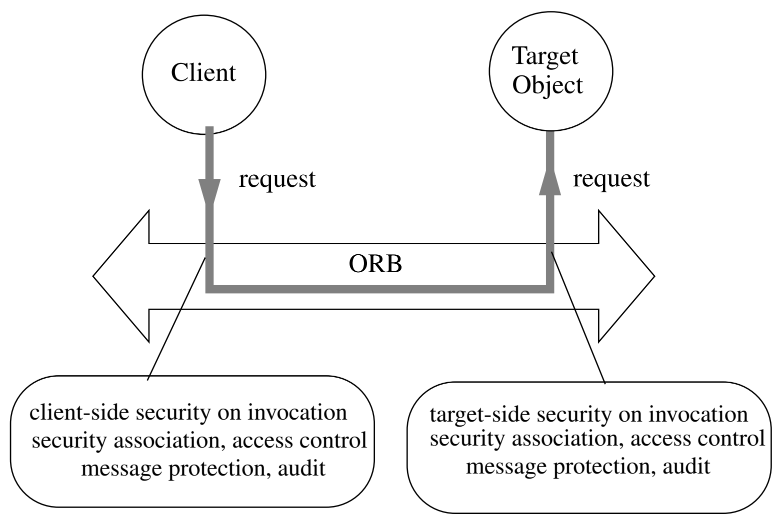


Figure 15-5 Invocation of Target Object via ORB

What security functionality is needed on object invocation depends on security policy. It may include:

- Establishing a security association between the client and target object so that each has the required trust that the other is who it claims to be. In many implementations, associations will normally persist for many interactions, not just a single invocation. (Within some environments, the trust may be achieved by local means, without use of authentication and cryptography.)
- Deciding whether this client (acting for this principal) can perform this operation on this object according to the access control policy, as described in Section 15.3.4, "Access Control Model," on page 15-24.

- Auditing this invocation if required, as described in Section 15.3.5, “Auditing,” on page 15-28.
- Protecting the request and response from modification or eavesdropping in transit, according to the specified quality of protection.

For all these actions, security functions may be needed at the client and target object sides of the invocation. For example, protecting a request may require integrity sealing of the message before sending it, and checking the seal at the target.

The association is asymmetric. If the target object invokes operations on the client, a new association is formed. It is possible for a client to have more than one association with the same target object. The application is unaware of security associations; it sees only requests and responses.

A secure system can also invoke objects in an insecure system. In this case, it will not be possible to establish trust between the systems, and the client system may restrict the requests passed to the target.

15.3.3.1 *Establishing Security Associations*

The client and target object establish a secure association by:

- Establishing trust in one another’s identities, which may involve the target authenticating the client’s security attributes and/or the client’s authenticating the target’s security name.
- Making the client’s credentials (including its security attributes) available to the target object.
- Establishing the security context which will be used when protecting requests and responses in transit between client and target object.

The way of establishing a security association between client and object depends on the security policies governing both the client and target object, whether they are in the same domain, and the underlying security mechanism. For example, the type of authentication and key distribution used.

The security policies define the choice of security association options such as whether one-way or mutual authentication is wanted between client and target, and the quality of protection of data in transit between them.

The security policy is enforced using underlying security mechanisms. This model allows a range of such mechanisms for security associations. For example, the mechanism may use symmetric (secret) key technology, asymmetric (public) key technology, or a combination of these. The Key Distribution services, Certification Authorities, and other underlying Security services, which may be used, are not visible in the model.

15.3.3.2 *Message Protection*

Requests and responses can be protected for:

- Integrity. This prevents undetected, unauthorized modification of messages and may detect whether messages are received in the correct order and if any messages have been added or removed.
- Confidentiality. This ensures that the messages have not been read in transit.

A security association may in some environments be able to provide integrity and confidentiality protection through mechanisms inherent in the environment, and so avoid having to use encryption.

The security policy specifies the strength of integrity and confidentiality protection needed. Achieving this integrity protection may require sealing the message and including sequence numbers. Confidentiality protection may require encrypting it.

This security reference model allows a choice of cryptographic algorithms for providing this protection.

Performing a request on a remote object using an ORB and associated services, such as TP, might cause a message to be constructed to send to the target as shown in Figure 15-6. At the target, this process is reversed, and results in the ORB invoking the operation on the target passing it the parameters sent by the client. The reply returned follows a similar path.

Message protection could be provided at different points in the message handling functionality of an ORB, which would affect how much of the message is protected.

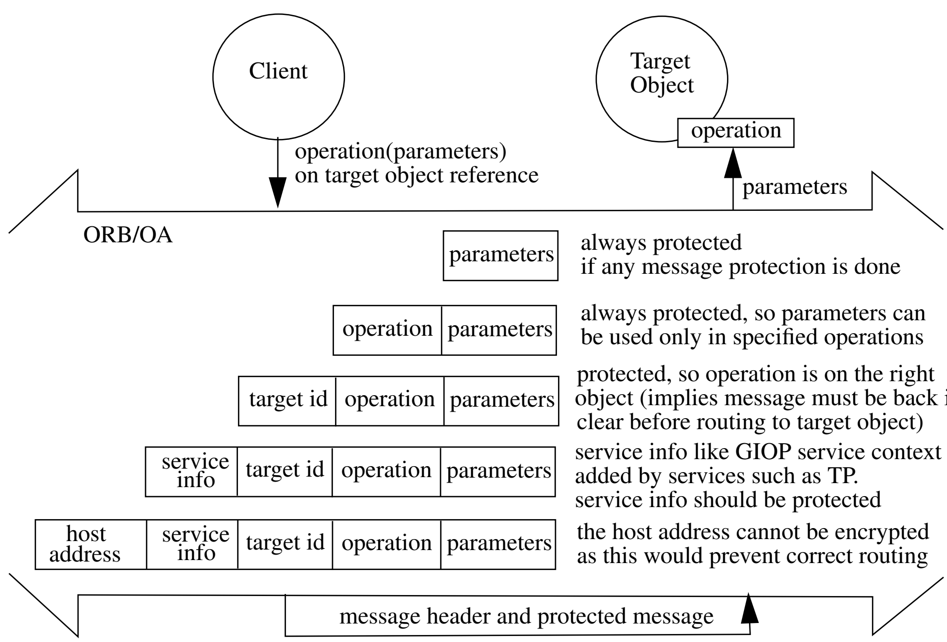


Figure 15-6 Message Protection

Messages are protected according to the quality of protection required which may be for integrity, but may also be for confidentiality. Both integrity and confidentiality protection are applied to the same part of the message. The request and response may be protected differently.

The CORBA security model can protect messages even when there is no security in the underlying communications software. In this case, the message protected by CORBA security includes the target id, operation, and parameters, and any service information included in the message.

In some systems, protection may be provided below the ORB message layer (for example, using the secure sockets layer or even more physical means). In this case, an ORB that knows such security is available will not need to provide its own message protection.

Note that as messages will normally be integrity-protected, this will limit the type of interoperability bridge that can be used. Any bridge that changes the protected part of the message after it has been integrity- (or confidentiality-) protected will cause the security check at the target to fail unless a suitable security gateway is used to re-protect the message.

15.3.4 Access Control Model

The model depicted in Figure 15-7 on page 15-25 provides a simple framework for many different access control security policies. This framework consists of two layers: an object invocation access policy, which is enforced automatically on object invocation, and an application access policy, which the application itself enforces.

The object invocation access policy governs whether this client, acting on behalf of the current principal, can invoke the requested operation on this target object. This policy is enforced by the ORB and the Security services it uses, for all applications, whether they are aware of security or not.

The application object access policy is enforced within the client and/or the object implementation. The policy can be concerned with controlling access to its internal functions and data, or applying further controls on object invocation.

All instantiations of the security reference model place at least some trust in the ORB to enforce the access policy. Even in architectures where the access control mediation occurs solely within the client and target objects, the ORB is still required to validate the request parameters and ensure message delivery as described above.

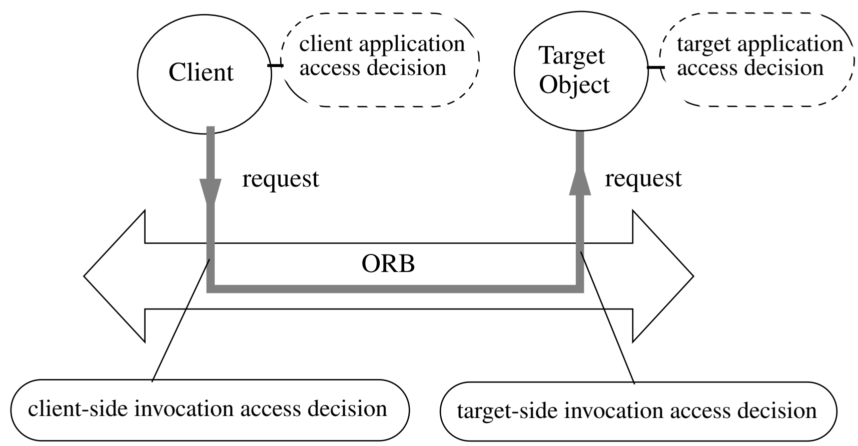


Figure 15-7 Access Control Model

The access control model shows the client invoking an operation as specified in the request, and also shows application access decisions, which can be independent of this.

15.3.4.1 Object Invocation Access Policy

A client may invoke an operation on the target object as specified in the request only if this is allowed by the object invocation access policy. This is enforced by Access Decision Functions.

Client-side access decision functions define the conditions that allow the client to invoke the specified operation on the target object. Target-side access decision functions define the conditions that allow the object to accept the invocation. One or both of these may not exist. Some systems may support target-side controls only, and even then, only use them for some of the objects.

The access policy for object invocation is built into these access decision functions, which just provide a yes/no answer when asked to check if access is allowed. A range of access policies can be supported as described in Section 15.5.9, "Access Control," on page 15-117.

The access decision function used on object invocation to decide whether access is allowed bases its decision on:

- The current privilege attributes of the principal (see Section 15.3.2, "Principals and Their Security Attributes," on page 15-20). Note that these can include capabilities.
- Any controls on these attributes, for example, the time for which they are valid.
- The operation to be performed.
- The control attributes of the target object (see Section 15.3.4, "Access Control Model," on page 15-24).

The first three of these functions are available as part of the environment of the object invocation.

The control attributes for the target object are associated with the object when it is created (though may be changed later, if security policy permits).

15.3.4.2 Application Access Policy

Applications may also enforce access policies. An application access policy may control who can invoke the application, extending the object invocation access policy enforced by the ORB, and taking into account other items such as the value of the parameters, or the data being accessed. As for standard object invocation access controls, there may be client and target object access decision functions.

An application object may also control access to finer-grained functions and data encapsulated within it, which are not separate objects.

In either case, the application will need its own access decision function to enforce the required access control rules.

15.3.4.3 Access Policies

The general access control model described here can be used to support a wide range of access policies including Access Control List schemes, label-based schemes, and capability schemes. This section describes the overall authorization model used for all types of access control.

The authorization model is based on the use of access decision functions, which decide whether an operation or function can be performed by applying access control rules using:

- Privilege attributes of the initiator (called initiator Access Control Information or ACI in ISO/IEC 10181-3).
- Control attributes of the target (sometimes known as the target ACI).
- Other relevant information about the action such as the operation and data, and about the context, such as the time.

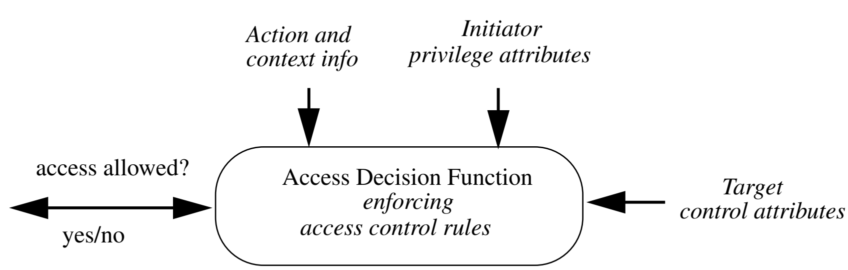


Figure 15-8 Authorization Model

The privilege and control attributes are the main variables used to control access; therefore, the following sections focus on these.

15.3.4.4 Privilege Attributes

A principal can have a variety of privilege attributes used for access control such as:

- The principal's access identity.
- Roles, which are often related to the user's job functions.
- Groups, which normally reflect organizational affiliations. A group could reflect the organizational hierarchy, for example, the department to which the user belongs, or a cross-organizational group, which has a common interest.
- Security clearance.
- Capabilities, which identify the target objects (or groups of objects), and their operations on which the principal is allowed.
- Other privileges that an enterprise defines as being useful for controlling access.

In an object system, which may be large, using individual identities for access control may be difficult if many sets of control attributes need to be changed when a user joins or leaves the organization or changes his job. Where possible, controls should be based on some grouping construct (such as a role or organizational group) for scalability.

The security reference model does not dictate the particular privilege attributes that any compliant secure system must support; however, this specification does define a standard, extensible set of privilege attribute types.

Note – In this specification, *privilege* is often used as shorthand for *privilege attribute.*

15.3.4.5 Control Attributes

Control attributes are associated with the target. Examples are:

- Access control lists, which identify permitted users by name or other privilege attributes, or
- Information used in label-based schemes, such as the classification of an object, which identifies (according to rules) the security clearance of principals allowed to perform particular operations on it.

An object system may have many objects, each of which may have many operations, so it may not be practical to associate control attributes with each operation on each object. This would impose too large an overhead on the administration of the system, and the amount of storage needed to hold the information.

Control attributes are therefore expected to be shared by categories of objects, particularly objects of the same type in the same security policy domain. However, they could be associated with an individual object.

Rights

Control attributes may be associated with a set of operations on an object, rather than each individual operation. Therefore, a user with specified privileges may have rights to invoke a specific set of operations.

It is possible to define what rights give access to what operations.

15.3.4.6 Access Policies Supported by This Specification

The model allows a range of access policies using control attributes, which can group subjects (using privileges), objects (using domains), and operations (using rights).

This specification defines a particular access policy type and associated management interface as part of security functionality Level 2. This is defined in DomainAccessPolicy Interface under Section 15.6.4, "Access Policies," on page 15-133.

Regardless of the access control policy management interface used (i.e., regardless of whether the particular Level 2 access policy interfaces or other interfaces not defined in this specification are used), all access decisions on object invocation are made via a standard access decision interface, so the access control policy can be changed either by administrative action on, or substitution of, the objects that define the policy and implement the access decision. However, different management interfaces will ordinarily be required for management of different types of control attributes.

15.3.5 Auditing

Security auditing assists in the detection of actual or attempted security violations. This is achieved by recording details of security-relevant events in the system. (Depending on implementation, recording an audit event may involve writing event information to a log, generating an alert or alarm, or some other action.) Audit policies specify which events should be audited under what circumstances.

There are two categories of audit policies:

- *system audit policies,* which control what events are recorded as the result of relevant system activities, and
- *application audit policies,* which control which events are audited by applications.

System events, which should be auditable, include events such as authentication of principals, changing privileges, success or failure of object invocation, and the administration of security policies. These system events may occur in the ORB or in security or other services, and these components generate the required audit records.

Application events may be security-relevant, and therefore may need auditing depending on the application. For example, an application that handles money transfers might audit who transferred how much money to whom.

Events can be categorized by event family (e.g., system, financial application service), and event type within that family. For example, there are defined event types for system events.

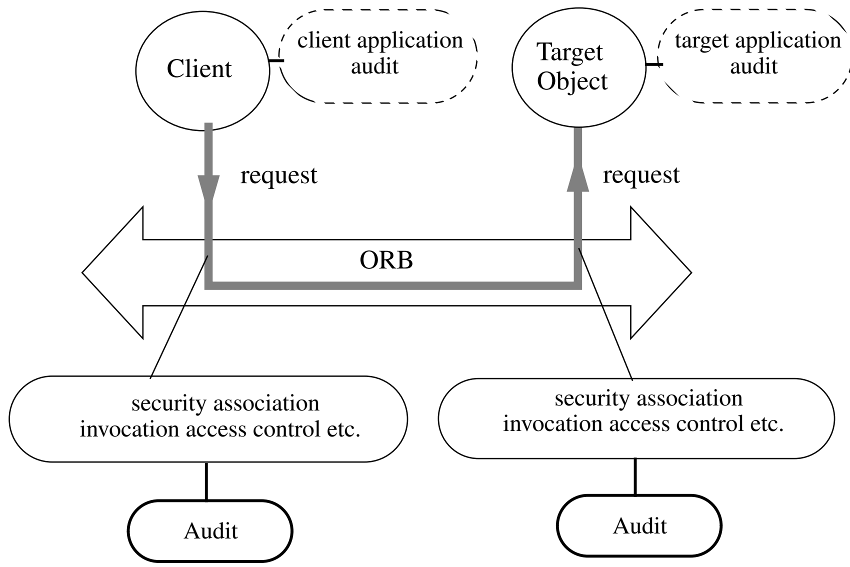


Figure 15-9 Auditing Model

Potentially a very large number of events could be recorded; audit policies are used to restrict what types of events to audit under which circumstances. System audit policies are enforced automatically for all applications, even security-unaware ones.

The invocation audit policy is enforced at a point in the ORB where the target object and operation for the request are known, and the reply status is known. The model supports audit policies where the decision on whether to audit an event can be based on the following:

- the event type (such as method invocation complete, access control check done, and security association made),
- the success or failure of this event (only failures may be audited),
- the object and the operation being invoked,
- the audit id of the principal on whose behalf the invocation is being done,
- and even the time of day.

This specification defines a particular invocation audit policy type and associated management interfaces as part of security functionality Level 2. This allows decisions on whether to audit an invocation to depend on the object type, operation, event type, and success or failure of this.

The specification also defines a particular audit policy type for application auditing, which allows decisions on whether to audit the event to be based on the event type and its success or failure.

Events can either be recorded on audit trails for later analysis or, if they are deemed to be serious, alarms can be sent to an administrator. Application audit trails may be separate from system ones. This specification includes how audit records are generated and then written to audit channels, but not how these records are filtered later, how audit trails and channels are kept secure, and how the records can be collected and analyzed.

15.3.6 Delegation

In an object system, a client calls on an object to perform an operation, but this object will often not complete the operation itself, so will call on other objects to do so. This will usually result in a chain of calls on other objects as shown in Figure 15-10.

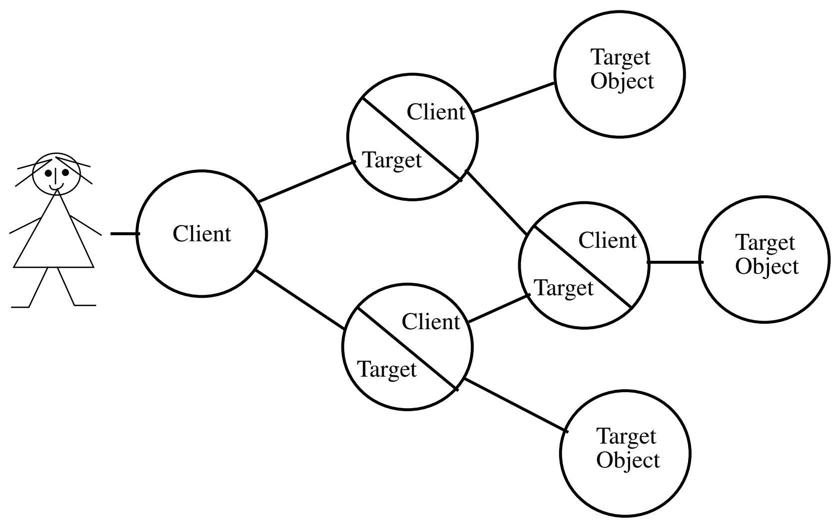


Figure 15-10 Delegation Model

This complicates the access model described in Section 15.3.4, “Access Control Model,” on page 15-24, as access decisions may need to be made at each point in the chain. Different authorization schemes require different access control information to be made available to check which objects in the chain can invoke which further operations on other objects.

In privilege delegation, the initiating principal’s access control information (i.e., its security attributes) may be delegated to further objects in the chain to give the recipient the rights to act on its behalf under specified circumstances.

Another authorization scheme is reference restriction where the rights to use an object under specified circumstances are passed as part of the object reference to the recipient. Reference restriction is not included in this specification, though described as a potential future security facility in Appendix F, “Facilities Not in This Specification” on page 15-358.

The following terms are used in describing delegation options:

- Initiator: the first client in a call chain.

- Final target: the final recipient in a call chain.
- Intermediate: an object in a call chain that is neither the initiator nor the final target.
- Immediate invoker: an object or client from which an object receives a call.

15.3.6.1 *Privilege Delegation*

In many cases, objects perform operations on behalf of the initiator of a chain of object invocations. In such cases, the initiator needs to delegate some or all of its privilege attributes to the intermediate objects which will act on its behalf.

Some intermediates in a chain may act on their own behalf (even if they have received delegated credentials) and perform operations on other objects using their own privileges. Such intermediates must be (or represent) principals so that they can obtain their own privileges to be transmitted to objects they invoke.

Some intermediates may need to use their own privileges at some times, and delegated privileges at other times.

A target may wish to restrict which of its operations an invoker can perform. This restriction may be based on the identity or other privilege attributes of the initiator. The target may also want to verify that the request comes from an authorized intermediate (or even check the whole chain of intermediates). In these cases, it must be possible to distinguish the privileges of the initiator and those of each intermediate.

Some restrictions may or may not be placed by the initiator about the set of objects which may be involved in a delegation chain.

When no restrictions are placed and only the initiator's privileges are being used, this case is called impersonation.

When restrictions are placed, additional information is used so that objects can verify whether or not their characteristics (e.g., their name or a part of their name) satisfy the restrictions. In order to allow clients or initiating objects to specify this additional information, objects can be (securely) associated with these characteristics (e.g., their name).

15.3.6.2 *Overview of Delegation Schemes*

There are potentially a large number of delegation models. They can all be captured using the following sentence.

An intermediate invoking a target object may perform:

1. one method on one object
2. several methods on one object
3. any method on:

	a. one object	
	b. some object(s)	(target restrictions)
	c. any object	(no target restrictions)

	(no privileges	
using	(a subset of the initiator's privileges	(simple delegation)
	(both the initiator's and its own privileges	(composite delegation)
		(combined or traced delegation,
	(received privileges and its own privileges	depending on whether privileges are combined or concatenated)
during some validity period		(part of time constraints)
for a specified number of invocations		(part of time constraints)

When delegating privileges through a chain of objects, the caller does not know which objects will be used in completing the request, and therefore cannot easily restrict privileges to particular methods on objects. It generally relies on the target's control attributes to do this.

A privilege delegation scheme may provide any of the other controls, though no one scheme is likely to provide all of them.

15.3.6.3 Facilities Potentially Available

Different facilities are available to intermediates (or clients) before initiating object invocations and to intermediate or target objects accepting an invocation.

Controls Used Before Initiating Object Invocations

A client or intermediate can specify restrictions on the use of the access control information provided to another intermediate or to a target object. Interfaces may allow support of the following facilities.

- Control of privileges delegated - An initiator (or an intermediate) can restrict which of its own privileges are delegated.
- Control of target restrictions - An initiator (or an intermediate) can restrict where individual privileges can be used. This restriction may apply to particular objects, or some grouping of objects. It may restrict the target objects, which may use some privileges for access control, and the intermediates, which can also delegate them.
- Control of privileges used - As previously described, there are several options for deciding which privileges an intermediate object may use when invoking another object. Note that delegated privileges are not actually delegated to a single target object; they are available to any object running under the same identity as the target object in the target object's address space (since any objects in the target's address space may retrieve the inbound Credentials and any object sharing the target's identity may successfully become the caller's delegate). The specified interfaces allow the following:
  - ***No delegation*** - The client permits the intermediate to use its privileges for access control decisions, but does not permit them to be delegated, so the intermediate object cannot use these privileges when invoking the next object in the chain.

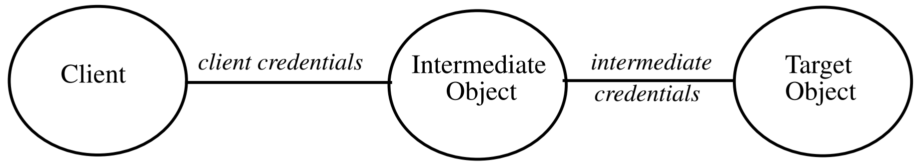


Figure 15-11 No Delegation

- ***Simple delegation*** - The client permits the intermediate to assume its privileges, both using them for access control decisions and delegating them to others. The target object receives only the client's privileges, and does not know who the intermediate is (when used without target restrictions, this is known as impersonation).

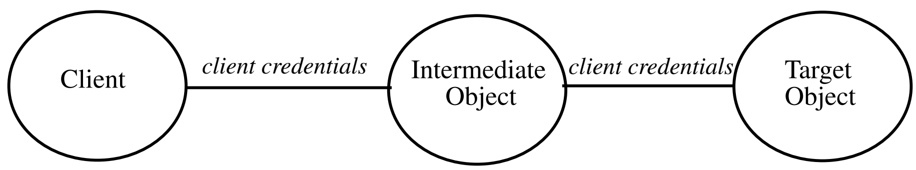


Figure 15-12 Simple Delegation

- ***Composite delegation*** - The client permits the intermediate object to use its credentials and delegate them. Both the client privileges and the immediate invoker’s privileges are passed to the target, so that both the client privileges and the privileges from the immediate source of the invocation can be individually checked.

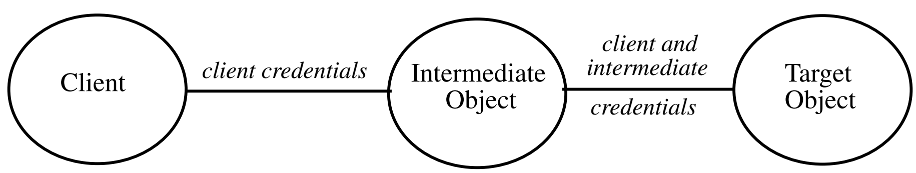


Figure 15-13 Composite Delegation

- ***Combined privileges delegation*** - The client permits the intermediate object to use its privileges. The intermediate converts these privileges into credentials and combines them with its own credentials. In that case, the target cannot distinguish which privileges come from which principal.

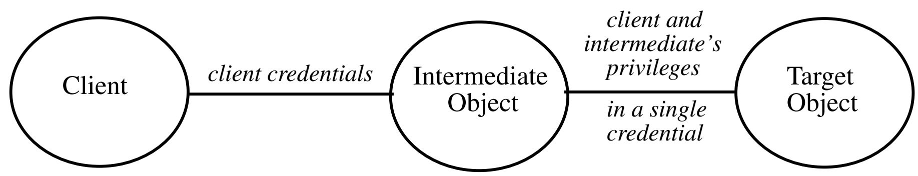


Figure 15-14 Combined Privileges Delegation

- ***Traced delegation*** - The client permits the intermediate object to use its privileges and delegate them. However, at each intermediate object in the chain, the intermediate's privileges are added to privileges propagated to provide a trace of the delegates in the chain.

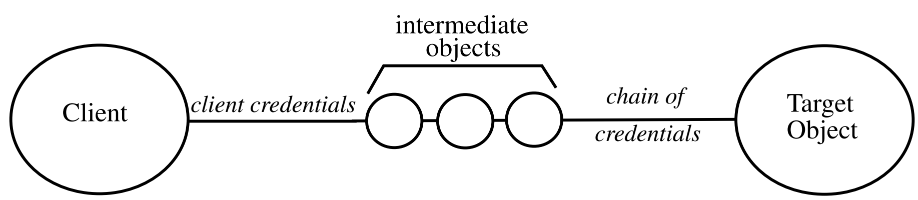


Figure 15-15 Traced Delegation

A client application may not see the difference between the last three options, it may just see them all as some form of “composite” delegation. However, the target object can obtain the credentials of intermediates and the initiator separately if they have been transmitted separately.

- ***Control of time restrictions*** - Time periods can be applied to restrict the duration of the delegation. In some implementations, the number of invocations may also be controllable.

Facilities Used on Accepting Object Invocations

An intermediate or a target object should be able to:

- Extract received privileges and use them in local access control decisions.
  Often only the privileges of the initiator are relevant. When this is not the case, only the privileges of the immediate invoker may be relevant. In some cases, both are relevant. Finally, the most complex authorization scheme may require the full tracing of the initiator and all the intermediates involved in a call chain.
  In addition, some targets may need to obtain the miscellaneous security attributes (such as audit identity, charging identity) and the associated target restrictions and time constraints.

- Extract credentials (when permitted) for use when making the next call as a delegate.

- Build (when permitted) new credentials from the received access control information with changed (normally reduced) privileges and/or different target restrictions or time constraints.

15.3.6.4 Specifying Delegation Options

The administrator may specify which delegation option should be used by default when an object acts as an intermediate. For example, he may specify whether a particular intermediate object normally delegates the initiating principal's privileges or uses its own, or both if needed. Also, the access policy used at the target could permit or deny access based on more than one of the privileges it received (e.g., the initiator's and the intermediate's). This allows many applications to be unaware of the delegation options in use, as many of the controls for delegation are done automatically by the ORB when the intermediate invokes the next object in the chain.

However, a security-aware intermediate object may itself specify what delegation it wants. For example, it may choose to use the original principal's privileges when invoking some objects and its own when invoking others.

15.3.6.5 Technology Support for Delegation Options

Different security technologies support different delegation models. Currently, no one security technology supports all the options described above.

In Security Functionality Level 1, all delegation is done automatically in the ORB according to delegation policy, so the objects in the chain cannot change the mode of delegation used, or restrict privileges passed and where or when they are used.

Of the options on which credentials are passed, only *no delegation* and *impersonation* (simple delegation without any target restrictions) *need* to be supported.

In Security Functionality Level 2, applications may use any of the interfaces specified, but may get a CORBA::NO_IMPLEMENT exception returned. Note that these interfaces do not allow the application to set controls such as target restrictions. Appendix F, "Facilities Not in This Specification" on page 15-358, includes potential future advanced delegation facilities, which include such controls.

15.3.7 Non-repudiation

Non-repudiation services provide facilities to make users and other principals accountable for their actions. Irrefutable evidence about a claimed event or action is generated and can be checked to provide proof of the action. It can also be stored in order to resolve later disputes about the occurrence or the nonoccurrence of the event or action.

The non-repudiation services specified here are under the control of the applications rather than used automatically on object invocation, so are only available to applications aware of this service.

Depending on the non-repudiation policy in effect, one or more pieces of evidence may be required to prove that some kind of event or action has taken place. The number and the characteristics of each depends upon that non-repudiation policy. As an example, evidence containing a timestamp from a trusted authority may be required to validate evidence.

There are many types of non-repudiation evidence, depending on the characteristics of the event or action. In order to distinguish between them, the types are defined and are part of the evidence. Conceptually, evidence may thus be seen as being composed of the following components:

- non-repudiation policy (or policies) applicable to the evidence
- type of action or event
- parameters related to the type of action or event

A date and time are also part of the evidence. This shows when an action or event took place and allows recovery from some situations such as the compromise of a key.

The evidence includes some proof of the origin of data, so a recipient can check where it came from. It also allows the integrity of the data to be verified.

Facilities included here allow an application to deal with evidence of a variety of types of actions or events. Two common types of non-repudiation evidence are the evidence of proof of creation of a message and proof of receipt of a message.

Non-repudiation of Creation protects against an originator's false denial of having created a message. It is achieved at the originator by constructing and generating evidence of Proof of Creation using non-repudiation services. This evidence may be sent to a recipient to verify who created the message, and can be stored and then made available for subsequent evidence retrieval.

Non-repudiation of Receipt protects against a recipient's false denial of having received a message (without necessarily seeing its content). It is achieved at the recipient by constructing and generating evidence of Proof of Receipt using the non-repudiation services. This is shown in Figure 15-16.

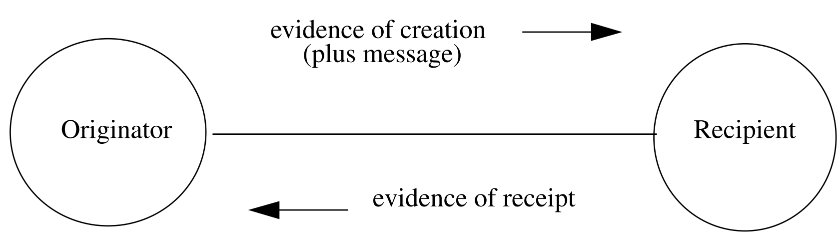


Figure 15-16 Proof of Receipt

One or more Trusted Third Parties need to be involved, depending on the choice of mechanism or policy.

Non-repudiation services may include:

- Facilities to generate evidence of an action and verify that evidence later.
- A delivery authority that delivers the evidence (often with the message) from the originator to the recipient. Such a delivery authority may generate *proof of origin* (to protect against a sender's false denial of sending a message or its content) and

proof of delivery (to protect against a recipient's false denial of having received a message or its content). Non-repudiation of Origin and Delivery are defined in ISO 7498-2.

- An evidence storage and retrieval facility used when a dispute arises. An adjudicator service may be required to settle the dispute, using the stored evidence.

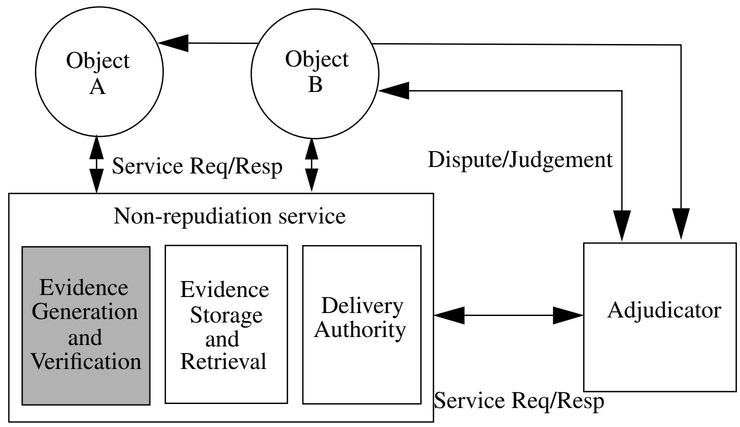


Figure 15-17 Non-repudiation Services

The non-repudiation services illustrated in Figure 15-17 are based on the ISO non-repudiation model. As the shaded box in the diagram indicates, this specification supports only Evidence Generation and Verification, which provides:

- Generation of evidence of an action.
- Verification of evidence of an action.
- Generation of a request for evidence related to a message sent to a recipient.
- Receipt of a request for evidence related to a received message.
- Analysis of details of evidence of an action.
- Collection of the evidence required for long term storage. In this case, more complete evidence may be needed.

The Non-repudiation Service allows an application to deal with a variety of types of evidence, not just the non-repudiation of creation and receipt previously described.

No Non-repudiation Evidence Delivery Authority is defined by this specification. It is anticipated that vendors will want to customize these authorities (which are responsible for delivering messages and related non-repudiation evidence securely in accordance with specific non-repudiation policies) to meet specialized market requirements. Also, no evidence storage and retrieval services are specified, as other object services can be used for this.

Note – This specification does not provide evidence that a request on an object was successfully carried out; it does not require use of non-repudiation within the ORB.

15.3.8 Domains

A domain (as specified in the ORB Interoperability Architecture) is a distinct scope, within which certain common characteristics are exhibited and common rules observed. There are several types of domains relevant to security:

- Security policy domain. The scope over which a security policy is enforced. There may be subdomains for different aspects of this policy.
- Security environment domain. The scope over which the enforcement of a policy may be achieved by some means local to that environment, so it does not need to be enforced within the object system. For example, messages will often not need cryptographic protection to achieve the required integrity when being transferred between objects in the same machine.
- Security technology domain. Where common security mechanisms are used to enforce the policies.

These can be independent of the ORB technology domains.

15.3.8.1 Security Policy Domains

A security policy domain is a set of objects to which a security policy applies for a set of security-related activities and is administered by a security authority. (Note that this is often just called a security domain.) The objects are the domain members. The policy represents the rules and criteria that constrain activities of the objects to make the domain secure. Security policies concern access control, authentication, secure object invocation, delegation, and accountability. An access control policy applies to the security policies themselves, controlling who may administer security-relevant policy information.

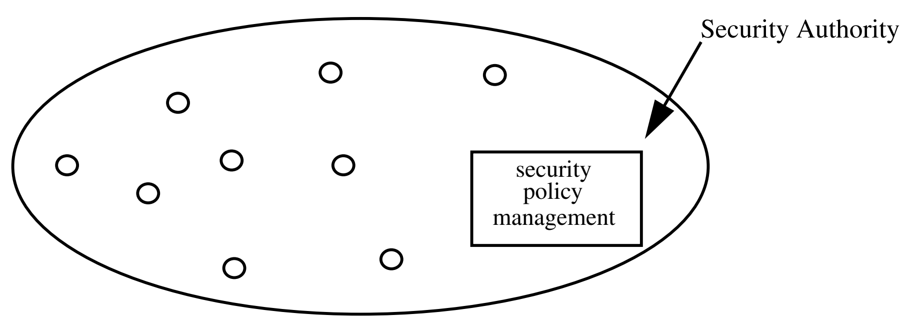


Figure 15-18 Security Policy Domains

Security policy domains provide leverage for dealing with the problem of scale in security policy management (by allowing application of policy at a domain granularity rather than at an individual object instance granularity).

Security policy domains permit application of security policy information to security-unaware objects without requiring changes to their interfaces (by associating the security policy management interfaces with the domain rather than with the objects to which policy is applied).

Domains provide a mechanism for delimiting the scope of administrators' authorities.

Policy Domain Hierarchies

A security authority must be identifiable and responsible for defining the policies to be applied to the domain, but may delegate that responsibility to a number of subauthorities, forming subdomains where the subordinate authorities' policies are applied.

Subdomains may reflect organizational subdivisions or the division of responsibility for different aspects of security. Typically, organization-related domains will form the higher-level superstructure, with the separation of different aspects of security forming a lower-level structure.

For example, there could be:

- An enterprise domain, which sets the security policy across the enterprise.
- Subdomains for different departments, each consistent with the enterprise policy but each specifying more specific security policies appropriate to that department.

With each department, authority may be further devolved:

- Authority for auditing could be the preserve of an audit administrator.
- Control of access to a set of objects could be the responsibility of a specific administrator for those objects.

This supports what is recognized as good security practice (it separates administrators' duties) while reflecting established organizational structures.

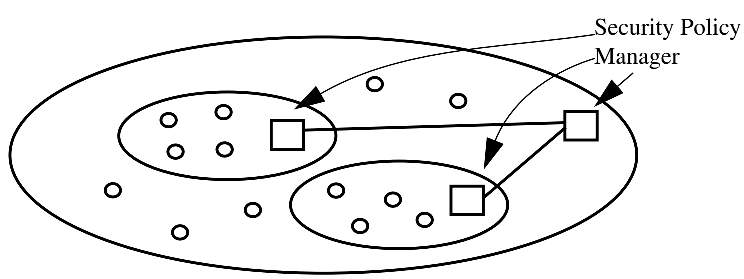


Figure 15-19 Policy Domain Hierarchies

Federated Policy Domains

As well as being structured into superior/subordinate relationships, security policy domains may also be federated. In a federation, each domain retains most of its authority while agreeing to afford the other limited rights. The federation agreement records:

- The rights given to both sides, such as the kind of access allowed.
- The trust each has in the other.

It includes an agreement as to how policy differences are handled, for example, the mapping of roles in one domain to roles in the other.

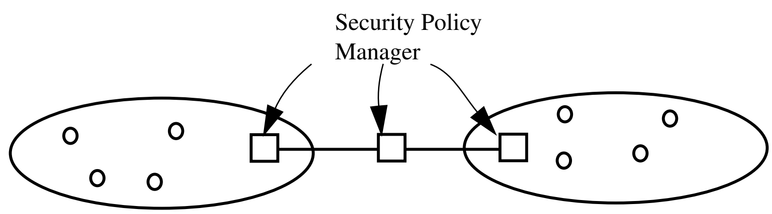


Figure 15-20 Federated Policy Domains

System- and Application-Enforced Policies

In a CORBA system, the "system" security policy is enforced by the distributed ORB and the Security services it uses and the underlying operating systems that support it. This is the only policy that applies to objects unaware of security.

The application security policy is enforced by application objects, which have their own security requirements. For example, they may want to control access to their own functions and data at a finer granularity than the system security policy provides.

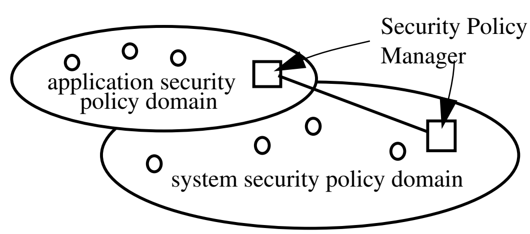


Figure 15-21 System- and Application-enforced Policies

Overlapping Policy Domains

Not all policies have the same scope. For example, an object may belong to one domain for access control and a different domain for auditing.

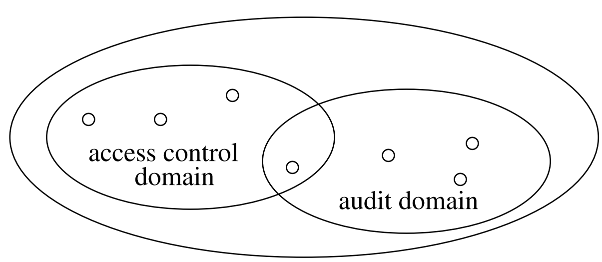


Figure 15-22 Overlapping Policy Domains

In some cases, there may even be overlapping policies of the same type (however, this specification does not require implementations to support overlapping policy domains of the same type).

15.3.8.2 Security Environment Domains

Security policy domains specify the scope over which a policy applies. Security environment domains are the scope over which the enforcement of the policies may be achieved by means local to the environment. The environment supporting the object system may provide the required security, and the objects within a specific environment domain may trust each other in certain ways. Environment domains are by definition implementation-specific, as different implementations run in different types of environments, which may have different security characteristics.

Environment domains are not visible to applications or Security services.

In an object system, the cost of using the security mechanisms to enforce security at the individual object level in all environments would often be prohibitive and unnecessary. For example:

- Preventing objects from interfering with each other might require them to execute in separate system processes or virtual machines (assuming the generation procedure could not ensure this protection) but, in most object systems, this would be considered an unacceptable overhead, if applied to each object.
- Authenticating every object individually could also impose too large an overhead, particularly where:
  - There is a large object population.
  - There is high connectivity, and therefore a large number of secure associations.
  - The object population is volatile, requiring objects to be frequently introduced to the Security services.

This cost can be reduced by identifying security environment domains where enforcement of one or more policies is not needed, as the environment provides adequate protection. Two types of environment domains are considered:

1. Message protection domains. These are domains where integrity and/or confidentiality is available by some specific means, for example, an underlying secure transport service is used. An ORB, which knows such protection exists, can exploit it, rather than provide its own message protection.
2. Identity domains. Objects in an identity domain can share the same identity. Objects in the same identity domain:
   - when invoking each other, do not need authentication to establish who they are communicating with.
   - are equally trusted by others to handle credentials received from a client. For example, if a client is prepared to delegate its rights to one object in the domain, it is prepared to delegate the same rights to all of them. If any object in the identity domain invokes a further object, that target object is prepared to trust the calling object based on the identity of its identity domain.

Note that neither of these affect what access controls apply to the object (except in that if trust is required and is not established with this domain, then access will be denied).

15.3.8.3 Security Technology Domains

These are domains that use the same security technology for enforcing the security policy. For example:

- The same methods are available for principal authentication and the same Authentication services are used.
- Data in transit is protected in the same way, using common key distribution technology with identical algorithms.
- The same types of access control are used. For example, a particular domain may provide discretionary access control using ACLs using the same type of identity and privilege attributes.
- The same audit services are used to collect audit records in a consistent way.

A particular security technology is normally used to authenticate principals and to form security associations between client and object and handle message protection. (Different technologies may be able to use the same privilege attributes, for example, the same access id and also the same audit id.) An important part of this is the security technology used for key distribution. There are two main types of security technology used for key distribution, both of which are available in commercial products:

- Symmetric key technology where a shared key is established using a trusted Key Distribution Service.
- Asymmetric (or "public") key technology where the client uses the public key of the target (certified by a Certification Authority), while the target uses a related private key.

Public key technology is also the most convenient technology upon which to implement non-repudiation, which has led to its use in several electronic mail products.

The CORBA security interfaces specified here are security-mechanism neutral, so they can be implemented using a wide variety of security mechanisms and protocols.

15.3.8.4 Domains and Interoperability

Interoperability between objects depends on whether they are in the same:

- Security technology domain
- ORB technology domain
- Security policy domains
- Naming and other domains

The level of security interoperability fully defined in this CORBA security specification is limited, though it includes an architecture that allows further interoperability to be added.

The following diagram shows a framework of domains and is used to discuss the interoperability goals of this specification.

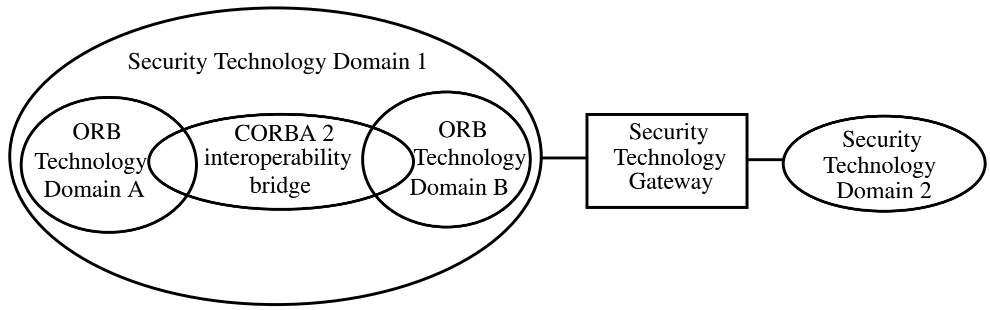


Figure 15-23 Framework of Domains

Interoperating between Security Technology Domains

Sending a message across the boundary between two different security technology domains is only possible if:

- The communication between the objects does not need to be protected, so security is not used between them, or
- A security technology gateway has been provided, which allows messages to pass between the two security technology domains. A gateway could be as simple as a physically secure link between the domains and an agreement between the administrators of the two domains to turn off security on messages sent over the link. On the other hand, it could be a very complicated affair including a protocol translation service with complicated key management logic, for example.

It is not a goal of this specification to define interoperability across Security Technology Domains, and hence to specify explicit support for security technology gateways. This is mainly because the technology is immature and appropriate common technology cannot yet be identified. However, where the security technology in the domains can support more than one security mechanism, this specification allows an appropriate matching mechanism to be identified and used.

Interoperating between ORB Technology Domains

If different ORB implementations are in the same security technology domain, they should be able to interoperate via a CORBA 2 interoperability bridge. However, there may still be restrictions on interoperability when:

- The objects are in different security policy domains, and the security attributes controlling policy in one domain are not understood or trusted in the other domain. As previously described, crossing a security policy boundary can be handled by a security policy federation agreement. This can be enforced in either domain or by a gateway.
- The ORBs are in different naming or other domains, and messages would normally be modified by bridges outside the trusted code of either ORB environment. Security protection prevents tampering with the messages (and therefore any

changes to object references in them). In general, crossing of such domains without using a Security Technology gateway is not possible if policy requires even integrity protection of messages.

15.3.9 Security Management and Administration

Security administration is concerned with managing the various types of domains and the objects within them.

15.3.9.1 Managing Security Policy Domains

For security policy domains, the following is required:

- Managing the domains themselves - creating and deleting them including controlling where they fit in the domain structure.
- Managing the members of the domain, including moving objects between domains.
- Managing the policies associated with the domains - setting details of the security policies as well as specifying which policies apply to which domains.

This specification focuses on management of the security policies. However, managing policy domains and their members in general are expected to be part of the Management Common Facilities, so only an outline specification is given here.

This specification includes a framework for the administering of security policies, and details of how to administer particular types of policy. For example, it includes operations to specify the default quality of protection for messages in this domain, the policy for delegating credentials, and the events to be audited.

General administration of all access control policies is not detailed, as the way of administering access control policies is dependent on the type of policy. For example, different administration is needed for ACL-based policies and label-based policies. However, the administration of the standard DomainAccessPolicy is defined.

Access policies may use *rights* to group operations for access control. Administration of the mapping of rights to operations is included in this specification. Such mapping of rights to operations is used by the standard DomainAccessPolicy, and can also be used by other access policies.

Interfaces for federation agreements allowing interaction with peer domains is left to a later security specification.

15.3.9.2 Managing Security Environment Domains

For environment domains, an administrator may have to specify the characteristics of the environment and which objects are members of the domain. This will often be done in an environment-specific way; therefore, no management interfaces for it are specified here.

15.3.9.3 Managing Security Technology Domains

For security technology domains, administration may include:

- Setting up and maintaining the underlying Security services required in the domain.
- Setting up and maintaining trust between domains in line with the agreements between their management.
- Administering entities in the way required by this security technology. Entities to be administered include principals, which have identities, long-term keys, and optionally privileged attributes.

Such administration is often security-technology-specific. Also, it may be done outside the object system, as it is a goal of this specification to allow common security technology to be used, and even allow a single user logon to object, as well as other applications. This specification does not include such security-technology-specific administration.

15.3.10 Implementing the Model

This reference model is sufficiently general to cover a very wide variety of security policies and application domains to allow conformant implementations to be provided to meet a wide variety of commercial and government secure systems in terms of both security functionality and assurance. (Any implementation of this model will need to identify the particular security policies it supports.)

The model also allows different ways of putting together the trusted core of a secure object system to address different requirements. There are a number of implementation choices on how to ensure that the security enforcement cannot be bypassed. This enforcement could be performed by hardware, the underlying operating system, the ORB core, or ORB services. Appendix D, "Guidelines for a Trustworthy System" on page 15-329 describes some of these options. (It is important when instantiating this architecture for a particular ORB product, or set of Security services supporting one or more ORBs, to identify what portions of the model must be trusted for what. This should be included in a conformance statement as described in Appendix E, "Conformance Statement" on page 15-352.)

15.4 Security Architecture

This section explains how the security model is implemented. It describes the complete architecture as needed to support all feature packages described in Section 15.2.2, "CORBA Security and Secure Interoperability Feature Packages," on page 15-12. Not all of these packages are mandatory for all implementors to support. See Appendix C, "Conformance Details" on page 15-315 for a definitive statement of conformance requirements.

This section starts by reviewing the different views that different users have of security in CORBA-compliant systems, as the security architecture must cater to these.

The structural model for security in CORBA-compliant systems is described. This includes some expansion of the ORB service concept introduced into CORBA 2 to support interoperability between ORBS.

The security object models for the three major views (application development, administration, and object system implementors) are then described.

15.4.1 Different Users' View of the Security Model

The security model can be viewed from the following users' perspectives:

- Enterprise management
- The end user
- The application developer
- Administration of an operational system
- The object system implementors

15.4.1.1 Enterprise Management View

Enterprise management is responsible for business assets including IT systems; therefore, they have the ultimate responsibility of protecting the information in the system. The enterprise view of security is mainly about protecting its assets against perceived threats at an affordable cost. This requires assessing the risks to the assets and the cost of countermeasures against them as described in Appendix E, Guidelines for a Trustworthy System. It will require setting a security policy for protecting the system, which the security administrators can implement and maintain.

Not all parts of an enterprise require the same type of protection of their assets. Enterprise management may identify different domains where different security policies should apply. Managers will need to agree how much they trust each other and what access they will provide to their assets. For example, when a user in domain A accesses objects in domain B, what rights should s/he have? One enterprise may also interwork with domains in other enterprises.

Enterprise management therefore knows about the structure of the organization and the security policies needed in different parts of it. Security policy options supported by the model include:

- A choice of access control policies. For example, controls can be based on job roles (or other attributes) and use ACL capabilities or label-based access controls.
- Different levels of auditing so that choosing which events to be logged can be flexibly chosen to meet the enterprise needs.
- Different levels of protection of information communicated between objects in a distributed system. For example, integrity only or integrity plus confidentiality.

The enterprise manager is not a direct user of the CORBA security system.

15.4.1.2 End User's View

The human user is an individual who is normally authenticated to the system to prove who he or she is.

The user may take on different job roles which allow use of different functions and data, thereby allowing access to different objects in the system. A user may also belong to one or more groups (within and across organizations) which again imply rights to access objects. A user may also have other privileges such as a security clearance that permits access to secret documents, or an authorization level that allows the user to authorize purchases of a given amount.

The user is modeled in the system as an initiating principal who can have privilege attributes such as roles and groups and others privileges valid to this organization.

Users invoke objects to perform business functions on their behalf, and their privilege attributes are used to decide what they can access. Their audit identity is used to make them individually accountable throughout the system. They have no idea of what further objects are required to perform the business function.

The user view is described further in the security model in Section 15.3, "Security Reference Model," on page 15-18.

15.4.1.3 Application Developer's View

The application developer is responsible for the business objects in the system: the applications. His main concern is the business functions to be performed.

Many application developers can be unaware of the security in the system, though their applications are protected by it. Therefore, much of the security in the system is hidden from the applications. ORB security services are called automatically on object invocation, and both protect the conversation between objects and control who can access them.

Some application objects need to enforce some security themselves. For example, an application might want to control access based on the value of the data and the time as well as the principal who initiated the operation. Also, an application may want to audit particular security-relevant activities.

The model includes a range of security facilities available for those applications that want to use them. For example:

- The quality of protection for object invocations can be specified and used to protect all communication with a particular target or just selected invocations.
- Audit can also be used independently of other security facilities and does not require the application to understand other security issues.
- Other functions, such as user authentication or handling privilege attributes for access control generally require more security understanding and operations on the objects, which represent the user in the system. However, this is still done via generic security interfaces, which hide the particular security technology used.

One special type of application developer is also catered for. The “application” that provides the user interface (user sponsor or logon client) needs an authentication interface capable of fitting with a range of authentication devices. However, the model also allows authentication to be done before calling the object system.

The application view is described in Section 15.5, “Application Developer’s Interfaces,” on page 15-88.

15.4.1.4 Administrator’s View

Administrators, like any other users, know about their job roles and other privileges, and expect these to control what they can do. In many systems, there will be a number of different administrators, each responsible for administering only part of the system. This may be partly to reduce the load on individual administrators, but partly for security reasons, for example to reduce the damage any one person can do.

Administrators and administrative applications see more of the system than other users or normal application developers. The application developers see individual objects whereas the administrator knows how these are grouped, for example, in policy domains.

In an operational system, administrators will be responsible for creating and maintaining the domains, specifying who should be members of the domain, its location, etc. They will also be responsible for administering the security policies that apply to objects in these domains.

An administrator may also be responsible for security attributes associated with initiating principals such as human users, though this may be done outside the object system. This would include administration of privilege attributes about users, but might also include other controls. For example, they might constrain the extent to which the user’s rights can be delegated.

The model does not include explicit management interfaces for managing domains or security attributes of initiating principals, though it does describe the resultant information. Note that the security facilities described here are also applicable to management. For example, management information needs to be protected from unauthorized access and protected for integrity in transit, and significant management actions, particularly those changing security information, need to be audited.

The administrator’s view is further described in Section 15.6, “Administrator’s Interfaces,” on page 15-130.

15.4.1.5 Object System Implementor’s View

Secure object system developers must put together:

- An ORB.
- Other Object Services and/or Common Facilities.
- The security services these require to provide the security features.

The system must be constructed in such a way as to make it secure.

The ORB implementor in a secure object system may use ORB Security services during object invocation, as defined in Section 15.4.2, “Structural Model,” on page 15-49. In addition, protection boundaries are required to prevent interference between objects and will need controlling by the ORB and associated Object Adapter and ORB services.

Certain interfaces are identified as Locality-Constrained. These interfaces are intended to be accessible only from within the context (e.g., process or RM-ODP capsule) in which they are instantiated (i.e., from within the protection boundary around that context). No object reference to these interfaces can therefore be passed meaningfully outside of that context. The exact details of how this protection boundary is implemented is an implementation detail that the implementor of the service will need to provide in order to establish that the implementation is secure. Locality- constrained objects may not be accessible through the DII/DSI facilities, and they may not appear in the Interface Repository. Any attempt to pass a reference to a locality- constrained object outside its locality, or any attempt to externalize it using **ORB::object_to_string** will result in the raising of the CORBA::NO_MARSHAL exception.

Object Service and Common Facilities developers may need to be security-aware if they have particular security requirements (for example, functions whose use should be limited or audited). However, like any application objects, most should depend on the ORB and associated services to provide security of object invocations.

The Security services implementor has to provide ORB Security services (for security of object invocations) and other security services to support applications’ view of security as previously defined. The ORB Security services implementor shares some application-visible security objects such as a principal’s credentials, and also sees the security objects used in making security associations. The Security services should use the Security Policy and other security objects defined in this model to decide what security to provide.

While these security objects may provide all the security required themselves, they will often call on external security services, so that consistent security can be provided for both object and other systems. The Security services defined in this specification are designed to allow for convenient implementation using generic APIs for accessing external security services so it is easier to link with a range of such services. Use of such external security services may imply use of existing, nonobject databases for users, certificates, etc. Such databases may be managed outside the object system.

The Implementor’s view is specified in Section 15.7, “Implementor’s Security Interfaces,” on page 15-156. The implications of constructing the system securely to meet threats are described in Appendix D, “Guidelines for a Trustworthy System” on page 15-329.

15.4.2 Structural Model

The architecture described in this section sets the major concepts on which the subsequent specifications are based.

The structural model has four major levels used during object invocation:

1. Application-level components, which may or may not be aware of security;
2. Components implementing the Security services, independently of any specific underlying security technology. (This specification allows the use of an isolating interface between this level and the security technology, allowing different security technologies to be accommodated within the architecture.) These components are:
   - The ORB core and the ORB services it uses.
   - Security services.
   - Policy objects used by these to enforce the Security Policy.
3. Components implementing specific security technology.
4. Basic protection and communication, generally provided by a combination of hardware and operating system mechanisms.

Figure 15-24 illustrates the major levels and components of the structural model, indicating the relationships between them. The basic path of a client invocation of an operation on a target object is shown.

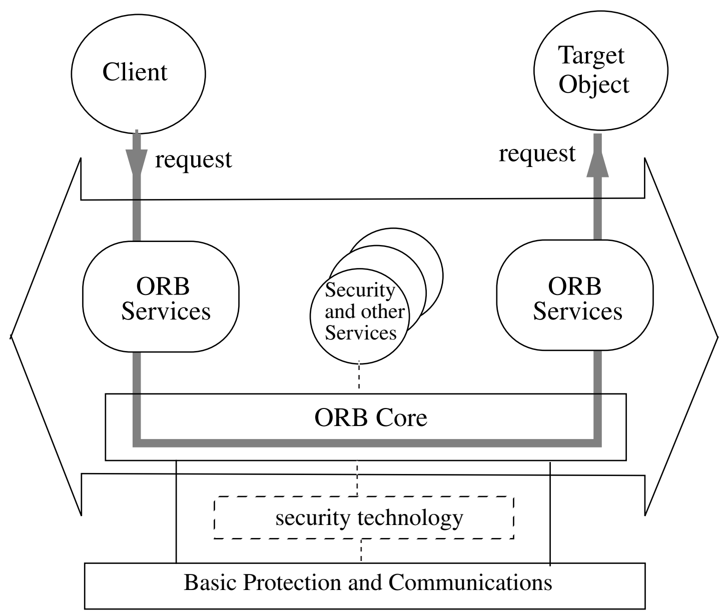


Figure 15-24 Structural Model

15.4.2.1 *Application Components*

Many application components are unaware of security and rely on the ORB to call the required security services during object invocation. However, some applications enforce their own security and therefore call on security services directly (see The Model as Seen by Applications, under Section 15.4.5, “Security Object Models,” on page 15-58). As in the OMA, the client may, or may not, be an object.

15.4.2.2 ORB Services

The ORB Core is defined in the CORBA architecture as "that part of the ORB that provides the basic representation of objects and the communication of requests." The ORB Core therefore supports the minimum functionality necessary to enable a client to invoke an operation on a target object, with the distribution transparencies required by the CORBA architecture.

An object request may be generated within an implicit context, which affects the way in which it is handled by the ORB, though *not* the way in which a client makes the request. The implicit context may include elements such as transaction identifiers, recovery data and, in particular, security context. All of these are associated with elements of functionality, termed ORB Services, additional to that of the ORB Core but, from the application view, logically present in the ORB.

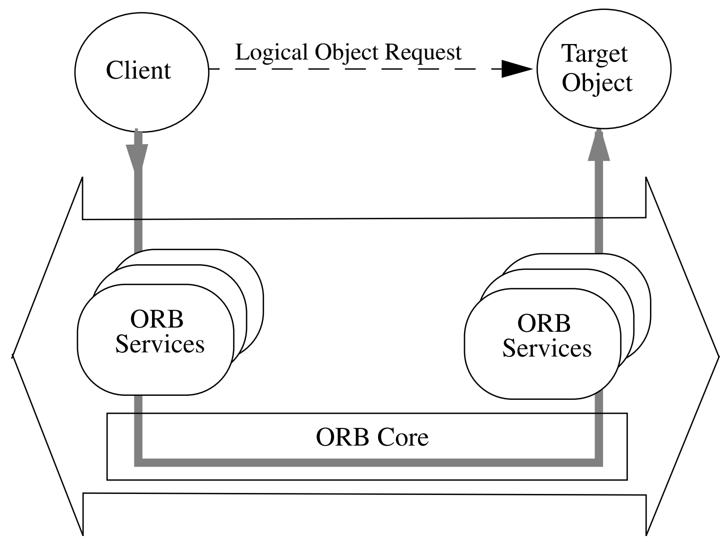


Figure 15-25 ORB Services

Selection of ORB Services

The ORB Services used to handle an object request are determined by:

- The security policies that apply to the client and target object because of the domains to which they belong, for example the access policies, default quality of protection.
- Other static properties of the client and target object such as the security mechanisms and protocols supported.
- Dynamic attributes, associated with a particular thread of activity or invocation; for example, whether a request has integrity or confidentiality requirements, or is transactional.

A client's ORB determines which ORB Services to use at the client when invoking operations on a target object. The target's ORB determines which ORB Services to use at the target. If one ORB does not support the full set of services required, then either the interaction cannot proceed or it can only do so with reduced facilities, which may be agreed to by a process of negotiation between ORBs.

Bindings and Object References at the Client

Before a client can use an object reference to invoke an operation of the target object in a secure way, a security association needs to be established associating the client to the target object, through the particular object reference. This security association is sometimes referred to as the binding. The creation and life-style of bindings are implicitly managed by the ORBs and hence the only invariant that one can depend on is that a binding is established before an invocation takes place.

The ORB determines how to establish the binding using the policies, static properties, and dynamic properties associated with the client and target. At the client, the client environment together with an object reference of the target object has associated with it, those policies and static properties of the target object (e.g., the quality of protection needed) that affect how the client's ORB establishes a binding to the object.

Associated with each binding is information specific to the particular usage by the client of the object reference. A binding is uniquely associated with:

- Each object reference of the target object that is held by the client.
- State information that is unique to the association between the target object and the client through the specific object reference (e.g., access policy domain, security context).
- An ORB instance in a process or capsule (c.f. RM-ODP[15]) in which the client is located.

A binding is distinct from the target object, though uniquely associated with it through the object reference. The lifetime of a binding is limited to that of the process or capsule that it is associated with, though it may be shorter (e.g., when the object reference to the target object is destroyed, the binding associated with the object reference is also destroyed).

There is state information associated with the binding at both the client and the server ends. This state information is local to the process or capsule in which the client and the server reside, and its lifetime is the same as that of the binding. The state associated with a binding is not accessible on the client side, since the implicitness of the binding and the uncertainty about its life-style makes such information of questionable value anyway. On the server side, some of this information is accessible through operations of the **Current** object.

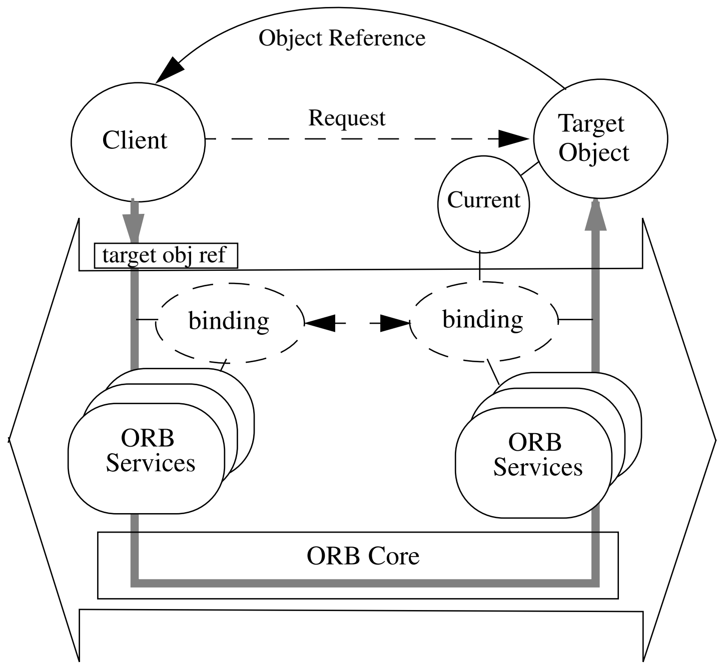


Figure 15-26 Object Reference

If a client requires that operations of the same target object be invoked with different invocation policies, it can do so by using the **Object::set_policy_overrides** operation to create new object references with the desired policies (that differ from those associated with the client's environment through the **Current** object) installed as overrides, and then use those new object references to carry out the invocations.

15.4.2.3 Security Services

In a secure object system, the ORB Services called will include ORB Security Services for secure invocation and access control.

ORB Security Services and applications may call on underlying security mechanisms for authentication, access control, audit, non-repudiation, and secure invocations. These security services form the Security Replaceability packages.

15.4.2.4 Security Policies and Domain Objects

A security policy domain is the set of objects to which common security policies apply as described in Security Policy Domains, under Section 15.3.8, "Domains," on page 15-38. The domain itself is not an object. However, there is a policy domain manager for each security policy domain. This domain manager is used when finding and managing the policies that apply to the domain. The ORB and security services use these to enforce the security policies relevant to object invocation.

When an object reference is created by the ORB, it implicitly associates the object reference with one or more Security Policy domains as described in Administrative Model, under Section 15.4.5, “Security Object Models,” on page 15-58. An implementation may allow object references to be moved between domains later. Since the only way to access objects is through object references, associating object references with policy domains and associated policies, implicitly associates the said policies with the object associated with the object reference. Care should be taken by the applications that are creating object references using POA operations (See the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*) to ensure that object references to the same object are not created by the server of that object with different domain associations.

There may be several security policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with each policy domain. (See “Administrative Model” on page 15-75, for a list of policy types.) These policy objects are shared between objects in the domain, rather than being associated with individual objects. (If an object needs to have an individual policy, then there must be a domain manager for it.)

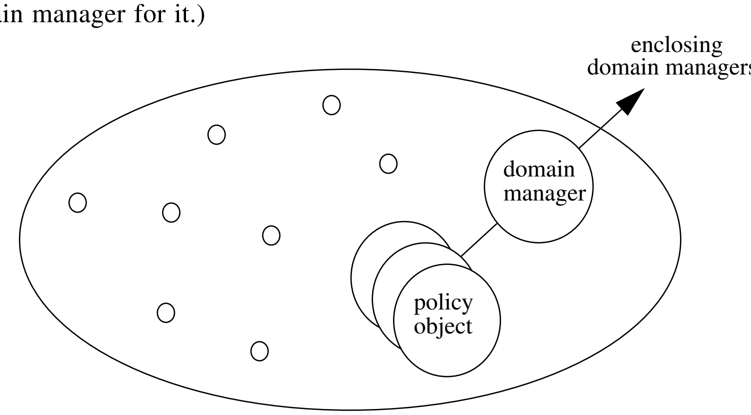


Figure 15-27 Domain Objects

Where an object reference is a member of more than one domain, for example, there is a hierarchy of domains, the object reference is governed by all policies of its enclosing domains. The domain manager can find the enclosing domain’s manager to see what policies it enforces.

The reference model allows an object reference to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

1. The *operational* interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own security policies. The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses

the policy object returned to enforce the policy (as described in "The Model as Seen by the Objects Implementing Security" on page 15-79). The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

2. The *administrative* interfaces used to set security policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so he is aware of the scope of what he is administering. (Administrative interfaces are described in "Administrative Model" on page 15-75.)

Applications will often not be aware of security at all, but will still be subject to security policy, as the ORB will enforce the policies for them. Security policy is enforced automatically by the ORB both when an object invokes another and when it creates another object.

An application that knows about security can also override certain default security policy details. For example, a client can override the default quality of protection of messages to increase protection for particular messages. (Application interfaces are described in "The Model as Seen by Applications" on page 15-58.)

Note – This specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them, moving objects between them, changing the domain structure and adding, changing, and removing policies applied to the domains. Such interfaces are expected to be the province of other object services and facilities.

15.4.3 Security Technology

The object security services previously described insulate the applications and ORBs from the security technology used. Security technology may be provided by existing security components. These do not have domain managers or objects. Security technology could be provided by the operating system. However, distributed, heterogeneous environments are increasingly being used, and for these, security technology is provided by a set of distributed security services. This architecture identifies a separate layer containing those components which actually implement the security services. It is envisaged that various technologies may be used to provide these and, furthermore, that a (set of) generic security interface(s) such as the GSS-API will be used to insulate the implementations of the security services from detailed knowledge of the underlying mechanisms. The range of services (and corresponding APIs) includes:

- The means of creating and handling the security information required to establish security associations, including keys.
- Message protection services providing confidentiality and integrity.

The use of standard, generic APIs for interactions with external security services not only allows interchangeability of security mechanisms, but also enables exploitation of existing, proven implementations of such mechanisms.

15.4.4 Basic Protection and Communications

15.4.4.1 Environment Domains

As described in "Security Environment Domains" on page 15-41, the way security policies are enforced can depend on the security of the environment in which the objects run. It may be possible to relax or even dispense with some security checks in the object system on interactions between objects in the same environment domain. For example, in a message protection domain where secure transport or lower layer communications is provided, encryption is not needed at the ORB level. In an identity domain, objects may share a security identity and so dispense with authenticating each other. Environment domains are implementation concepts; they do not have domain managers.

Environment domains can be exploited to optimize performance and resource usage.

15.4.4.2 Component Protection

The maintenance of integrity and confidentiality in a secure object system depends on proper segregation of the objects, which may include the segregation of security services from other components. At the lowest level of this architecture, Protection Domains, supported by a combination of hardware and software, provide a means of protecting application components from each other, as well as protecting the components that support security services. Protection Domains can be provided by various techniques, including physical, temporal, and logical separation.

The Security Architecture identifies various security services, which mediate interactions between application-level components: clients and target objects. The Security Object Models show how these mechanisms can themselves be modeled and implemented in terms of additional objects. However, security services can only be effective if there is some means of ensuring that they are always invoked as required by security policies: it must be possible to guarantee, to any required level of assurance, that applications cannot bypass them. Moreover, security services themselves, like other components, must be subject to security policies.

The general approach is to establish protection boundaries around groups of one or more components which are said to belong to a protection domain. Components belonging to a protection domain are assumed to trust each other, and interactions between them need not be mediated by security services, whereas interactions across boundaries may be subject to controls. In addition, it is necessary to provide a means of establishing a trust relationship between components, allowing them to interact across protection boundaries, in a controlled way, mediated by security services.

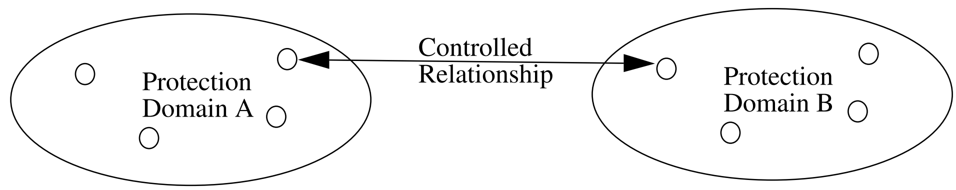


Figure 15-28 Controlled Relationship

In this architecture, the trusted components supporting security services are encapsulated by objects, as described in "The Model as Seen by the Objects Implementing Security" on page 15-79. Clearly, objects that encapsulate sensitive security information must be protected to ensure that they can only be accessed in an appropriate way.

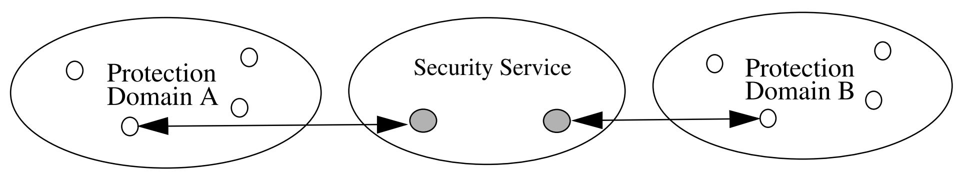


Figure 15-29 Object Encapsulation

Protection boundaries and the controlled relationships that cross those boundaries must inevitably be supported by functionality more fundamental than that of the Security Object Models, and invariably requires a combination of hardware and operating system mechanisms. Whichever way it is provided, this functionality constitutes part of the Trusted Computing Base.

Protection boundaries may be created by physical separation, interprocess boundaries, or within process access control mechanisms (e.g., multilevel "onionskin" hardware-supported access control). Less rigorous protection may be acceptable in some circumstances, and in such cases protection boundaries can be provided, for example, by using appropriate compilation tools to conceal protected interfaces and data.

The architecture is defined in a modular way so that, where necessary, it is possible for implementations to create protection boundaries between:

- Application components, which do not trust each other.
- Components supporting security services and other components.
- Components supporting security services and each other.

In addition, controlled communication across protection boundaries may be required. In such cases, it must be possible to constrain components within a protection boundary to interact with components outside the protection boundary only via controlled communications paths (it must not be possible to use alternative paths). Such communication may take many forms, ranging from explicit message passing to implicit sharing of memory.

15.4.5 Security Object Models

This section describes the objects required to provide security in a secure CORBA system from three viewpoints:

1. The model as seen by applications.
2. The model as seen by administrators and administrative applications.
3. The model as seen by the objects implementing the secure object system.

For each viewpoint, the model describes the objects and the relationships between them, and outlines the operations they support. A summary of all objects is also given.

15.4.5.1 The Model as Seen by Applications

Many applications in a secure CORBA system are unaware of security, and therefore do not call on the security interfaces. This subsection is therefore mainly relevant to those applications that are aware of and utilize security. Facilities available to such applications are:

- Finding what security features this implementation supports.
- Establishing a principal's credentials for using the system. Authenticating the principal may be necessary.
- Selecting various security attributes (particularly privileges) to affect later invocations and access decisions.
- Making a secure invocation.
- Handling security at a target object and at intermediates in a chain of objects, including use of credentials for application control of access and delegation.
- Auditing application activities.
- Non-repudiation facility -- generation and verification of evidence so that actions cannot be repudiated.
- Finding the security policies that apply to this object.

The Security Service interfaces that are available to the application writer are primarily found in the **SecurityLevel1**, **SecurityLevel2**, **NRservice**, and **SecurityAdmin** modules.

Finding Security Features

An application can find out what security features are supported by this secure object implementation. It does this by calling on the ORB to **get_service_information**. Information returned includes the security functionality level and options supported and the version of the security specification to which it conforms. It also includes security mechanisms supported (though the ORB Security Services, rather than applications, need this).

Establishing Credentials

If the principal has already been authenticated outside the object system, then **Credentials** can be obtained from **Current**.

If the principal has not been authenticated, but is only going to use public services which do not require presentation of authenticated privileges, a **Credentials** object may be created without any authenticated principal information.

If the principal has not been authenticated, but is going to use services that need him to be, then authentication is needed as shown in Figure 15-30.

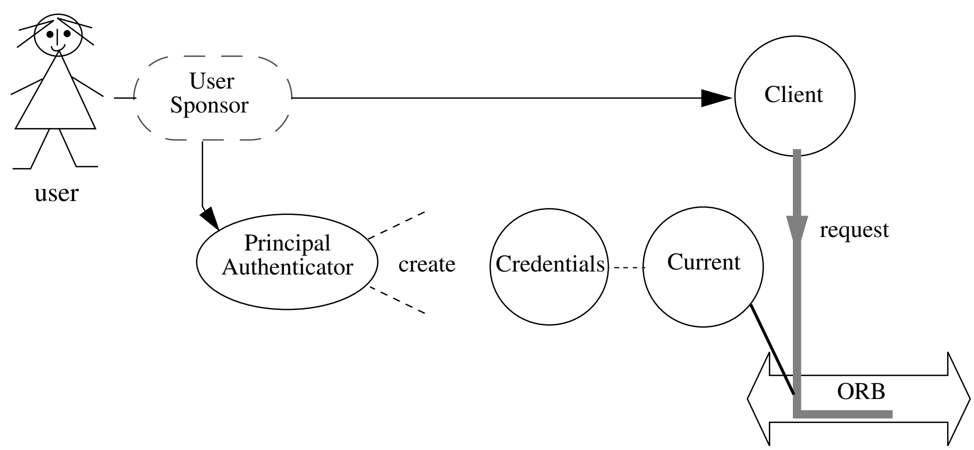


Figure 15-30 Authentication

User sponsor

The user sponsor is the code that calls the CORBA Security interfaces for user authentication. It need not be an object, and no interface to it is defined. It is described here so that the process of **Credentials** acquisition may be understood.

The user provides identity and authentication data (such as a password) to the user sponsor, and this calls on the **Principal Authenticator** object, which authenticates the principal (in this case, the user) and obtains **Credentials** for it containing authenticated identity and privileges.

The user sponsor represents the entry point for the user into the secure system. It may have been activated, and have authenticated the user, before any client application is loaded. This allows unmodified, security-unaware client applications to have **Credentials** established transparently, prior to making invocations.

There is no concept of a target object sponsor.

Principal Authenticator

The **Principal Authenticator** object is the application-visible object responsible for the creation of **Credentials** for a given principal. This is achieved in one of two ways. If the principal is to be authenticated within the object system, the user sponsor invokes the **authenticate** operation of the **Principal Authenticator** object (and **continue_authentication** if needed for multiexchange authentication dialogues).

Credentials

A **Credentials** object holds the security attributes of a principal. These security attributes include its authenticated (or unauthenticated) identities and privileges and information for establishing security associations. It provides operations to obtain and set security attributes of the principal it represents.

There may be credentials for more than one principal, for example, the initiating principal who requested some action and the principal for the current active object. **Credentials** are used on invocations and for non-repudiation.

There is an **is_valid** operation to check if the credentials are valid and a **refresh** operation to refresh the credentials if possible.

Current

The **Current** object represents the current execution context at both client (both for object or non-object clients) and target objects. In a secure environment, the interfaces **SecurityLevel1::Current** which is derived from **CORBA::Current** and **SecurityLevel2::Current** which is derived from **SecurityLevel1::Current**, give access to security information associated with the execution context. **Current** gives access to the **Credentials** associated with the execution environment. Object invocations use **Credentials** in **Current**, unless they have been overridden, by a security-aware client, in the specific object reference being used for the invocation. If a user sponsor is used, it should set the user's credentials for subsequent invocations in **Current**. This may also be done as the result of initializing the ORB when the user has been authenticated outside the object system. This allows a security-unaware application to utilize the credentials without having to perform any explicit operation on them.

At target and intermediate objects, other **Credentials** are also available via **Current**.

Handling Multiple Credentials

An application object may use different **Credentials** with different security characteristics for different activities.

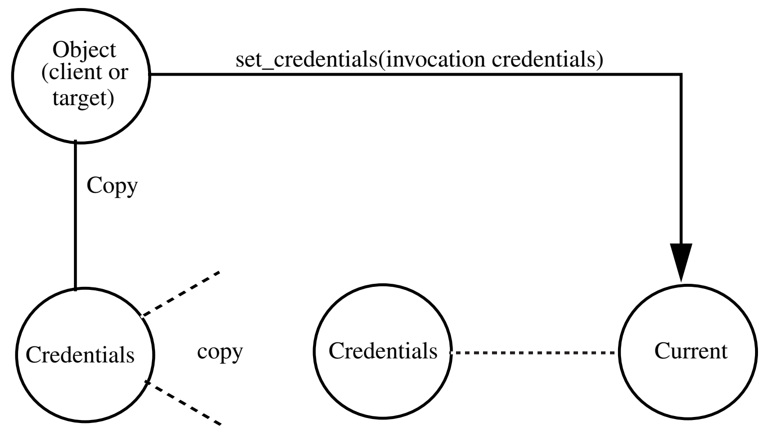


Figure 15-31 Multiple Credentials

The **Credentials::copy** operation can be used to make a copy of the **Credentials** object. The new **Credentials** object (i.e., the copy) can then be modified as necessary, using its interface, before it is used in an invocation.

When all required changes have been made, the **Current::set_credentials** operation can be used to specify a different **Credentials** object as the default for subsequent invocations.

At any stage, a client or target object can find the default credentials for subsequent invocations by calling **Current::get_credentials**, asking for the invocation credentials. These default credentials will be used in all invocations using object references in which the invocation credentials have not been overridden.

Selecting Security Attributes

A client may require different security for different purposes, for example, to enforce a least privilege policy and so specify that limited privileges should be used when calling particular objects, or collections of objects, and restrict the scope to which these privileges are propagated. A client may also want to protect conversations with different targets differently.

There are two ways to change security attributes for a principal:

1. Setting attributes on the credentials for that principal. If attributes are set on the credentials, these apply to subsequent object invocations using those credentials. It can therefore apply to invocations of many target objects.

2. Overriding attributes on the target object reference. Attributes thus set apply to subsequent invocations, which this client makes using this reference.

In both cases, the change applies immediately to further object invocations associated with these credentials or this object reference.

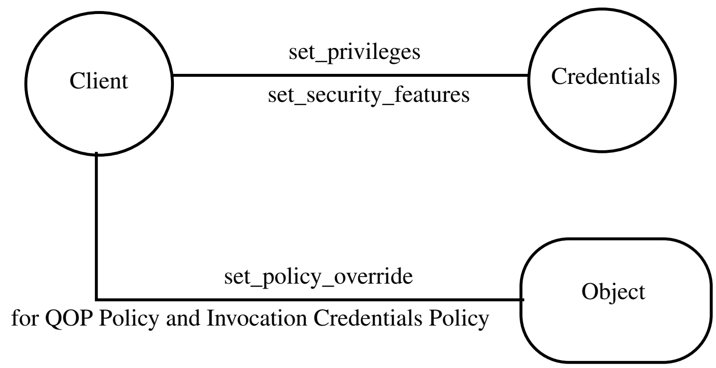


Figure 15-32 Changing Security Attributes

A wider range of attributes can be set on the credentials than on a specific object reference. Operations available include:

- **set_privileges** to set privileges in the credentials. The system will reject an attempt to set privileges if the calling principal is not entitled to one or more of the requested privileges. There may be additional restrictions on which privileges may be claimed if the caller is an intermediate in a delegated call chain attempting to set privileges on delegated **Credentials**.
- **set_security_features** to set such features as the quality of protection of messages (and the credentials to use for future invocations when at an intermediate object).

Setting any of these attributes may result in a new security association being needed between this client and target.

Note – This specification does not contain an operation to restrict when and where these privileges can be used in target objects or delegated, though this may be specified in the future (see Appendix Section F.12, "Target Control of Message Protection," on page 15-362).

A client may want to use different privileges or controls when invoking different targets. It can do this by obtaining a new object reference using the **set_policy_overrides** specifying the invocation credentials policy to be used with that target, and then use the object reference thus obtained to carry out the invocation.

A client may want to specify that a particular quality of protection applies only to selected invocations of a target object. For example, it may want confidentiality of selected messages. The client can do this by using **set_policy_overrides,** specifying a **QOP Policy** on the new object reference. It can continue to use the original object reference for those invocations where confidentiality is no longer required.

The **set_policy_overrides** operation returns a new object reference to the same target object as the one on which this operation is invoked. This new reference has the policy overrides set in it. Any invocations through this new reference will use the overrides set in the reference. The creation of this newly annotated object reference has no effect on the target object.

Equivalent **get_** operations are also provided to permit an application to determine the security-specific options currently requested, for example **get_attributes** (privileges, and other attributes such as audit id) and **get_security_features** on credentials objects.

The *security features-, invocation credentials-, qop-,* and *mechanism*-related policies that are in effect on a given object reference can be obtained by using the **get_policy** operation asking for the appropriate type of policy object.

Making a Secure Invocation

A secure invocation is made in the same way as any other object invocation, but the actual invocation is mediated by the ORB Security Services, invisibly to the application, which enforce the security requirements, both in terms of policy and application preference. The following diagram shows an application making the invocation, and the ORB Security Services utilizing the security information in **Current**, and hence the **Credentials** there.

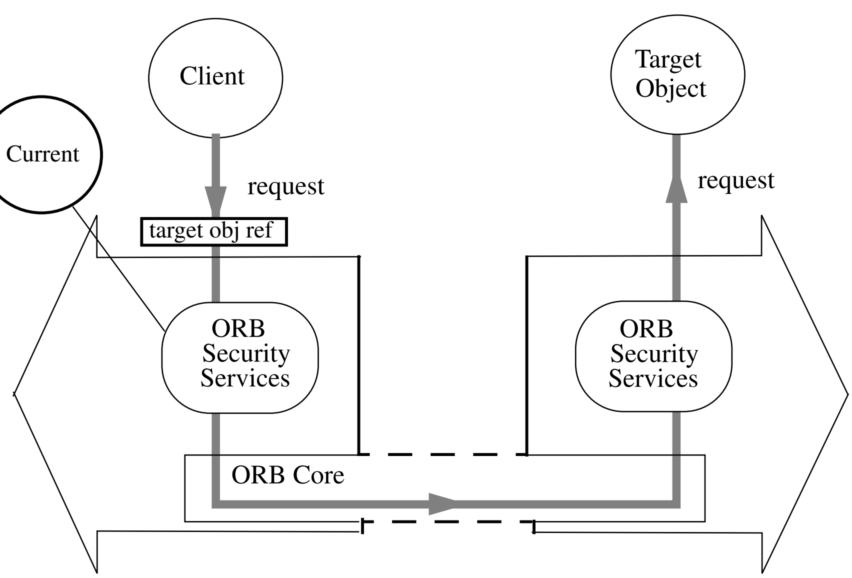


Figure 15-33 Making a Secure Invocation

Note – For any given invocation, it is target and client security policy that determines which (if any) ORB Security Services mediate that invocation. If the policy for a given invocation requires no security, then no services will be used. Similarly, if only access control is required, then only the ORB Security Service responsible for the provision of access control will be invoked.

Security at the Target

At the target, as at the client, the **Current** object is the representative of the local execution context within which the target object's code is executing. The **Current** object can be used by the target object, or by ORB and Object Service code in the target object's execution context, to obtain security information about an incoming security association and the principal on whose behalf the invocation was made.

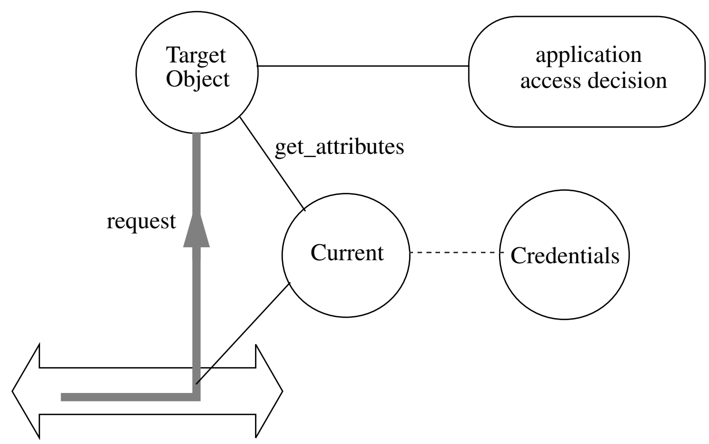


Figure 15-34 Target Object Security

A security-aware target application may obtain information about the attributes of the principal responsible for the request by invoking the **Current::get_attributes** operation. The target normally uses **get_attributes** to obtain the privilege attributes it needs to make its own access decisions.

When **Current::get_attributes** is invoked from the target object it returns the attributes from the incoming **Credentials** from the client. When **Current::get_attributes** is invoked by a client the attributes from the **Credentials** of the user (e.g., the one that was created by the **PrincipalAuthenticator**) is returned. Invoking **Credentials::get_attribute** always returns the attributes contained in that **Credentials** object.

Intermediate Objects in a Chain of Objects

When a client calls a target object to perform some operation, this target object often calls another object to perform some function, which calls another object and so on. Each intermediate object in such a chain acts first as a target, and then as a client, as shown in Figure 15-35 on page 15-65.

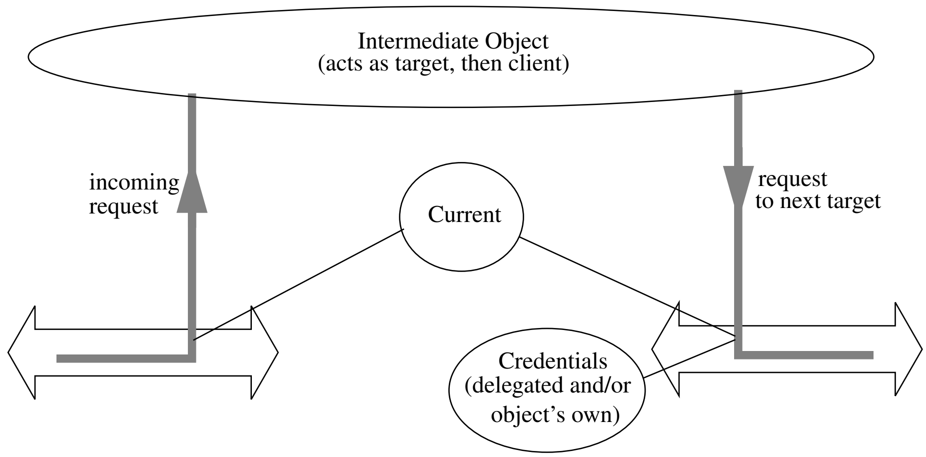


Figure 15-35 Security-unaware Intermediate Object

For a security-unaware intermediate object, **Current** has a reference to the security context established with the incoming client. When this intermediate object invokes another target, either the delegated credentials from the client or the credentials for the intermediate object's principal (or both) become the current ones for the invocation. The security policy for this intermediate object governs which credentials to use, and the ORB Security Services enforce the policy, passing the required credentials to the target, subject to any delegation constraints. The intermediate object's principal will be authenticated, if needed, by the ORB Security Services.

A security-aware intermediate object can:

- Use the privileges of any delegated credentials for access control.
- Decide which credentials to use when invoking further targets.
- Restrict the privileges available via these credentials to further clients (where security technology permits).

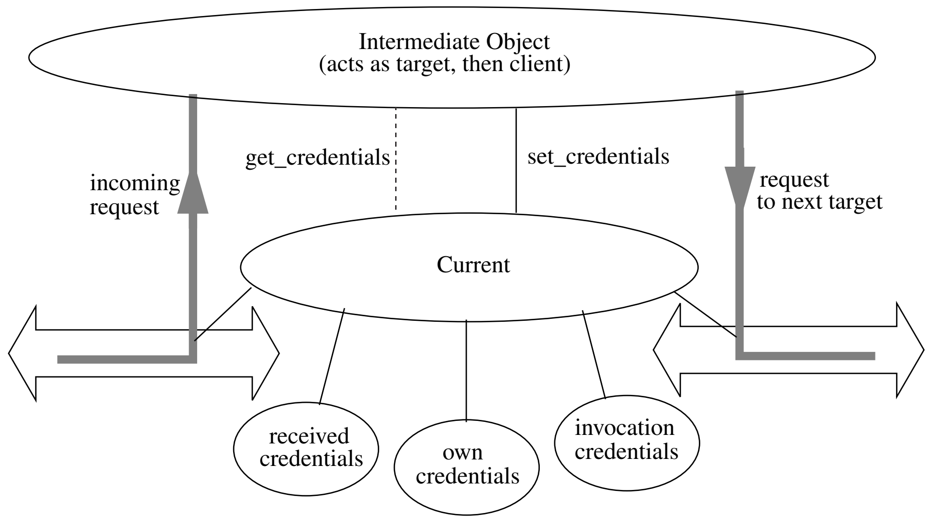


Figure 15-36 Security-aware Intermediate Object

After a chain of object calls, the target can call **Current::get_attributes** as previously described. Note that this call always obtains the privilege and other attributes associated with the first of the received credentials.

The target can use the **received_credentials** attribute of **Current** to get the incoming credentials. This may be a list of one or more credential objects depending on the authentication and delegation technology used. If more than one credential is returned, the first credential is that of the initiator. Other credentials are of intermediates in the chain. After composite delegation (see Section 15.3.6, "Delegation," on page 15-30), the credentials are of the initiator and immediate invoker. After traced delegation, credentials for all intermediates in the chain will be present (as well as the initiator). If a target object receives a request which includes credentials for more than one principal, it may choose which privileges to use for access control and which credentials to delegate, subject to policy.

An intermediate object may wish to make a copy of the incoming credentials, modify and then delegate them, though not all implementations will support this modification. In this case, it must acquire a reference to the incoming credentials (using the **received_credentials** attribute of **Current**), and then use **Credentials::set_privileges** to modify them. Finally it can call **Current::set_credentials** to make the received credentials the default ones for subsequent invocations. When the **received_credentials** are passed to **set_credentials**, whether it is a delegation or not needs to be specified to the **set_credentials** operation, and it takes appropriate action.

If the intermediate object wishes to change the association security defaults (for example, the quality of protection) for subsequent invocations to a specific target object, it can do so by using the **Object::set_policy_overrides** operation to create a copy of the object reference to the target with the required QOP set as override in the object reference thus obtained. The overridden QOP will apply to subsequent invocations through this new reference.

The intermediate object may be a principal and wish to use its own identity and some specific privileges in further invocations, rather than delegating the ones received. In this case, it can call **authenticate** operation of the **PrincipalAuthenticator** to obtain the appropriate credential, and then call **Credentials::set_privileges** to establish the appropriate rights. After doing this, it can use **Current::set_credentials** to establish its credential as the default for future invocations.

If the intermediate does not have its own individual **Credential** object (for example, as it does not have an individual security name) but instead shares credentials with other objects, it can use the **own_credentials** attribute of **Current** to get a copy of the **Credentials** (which will have been set up automatically). It can then do a **Credentials::copy** and then a **Credentials::set_privileges,** etc. on these as appropriate, and then use it to obtain a new object reference for the object it intends to invoke, with invocation credentials policy overridden using the **Credentials** constructed above.

If it wants to use composite delegation with a modified version of its own credentials, it should call **Current::set_credentials** (specifying its own credentials) and the required delegation mode before making the invocation. Note that this will not modify the credentials shared with other objects.

Security Mechanisms

Applications are normally aware of the security mechanism used to secure invocations. The secure object system is aware of the mechanisms available to both client and target object and can choose an acceptable mechanism. However, some security-sophisticated applications may need to know about, or even control the choice of mechanisms. They can get information on the mechanism policy currently in effect by using the **get_policy** operation of the object reference. They can do invocations using a different mechanism from the default by using the **set_policy_overrides** operation of the object reference to obtain a new object reference with the desired mechanism policy set as override in it and use it for invocations that need the new mechanism.

Application Access Policies

Applications can enforce their own access policies. No standard application access policy is defined, as different applications are likely to want different criteria for deciding whether access is permitted. For example, an application may want to take into account data values such as the amount of money involved in a funds transfer.

However, it is recommended that the application use an access decision object similar to the one used for the invocation access policy. This is to isolate the application from details of the policy. Therefore, the application should decide if access is needed as shown in Figure 15-37.

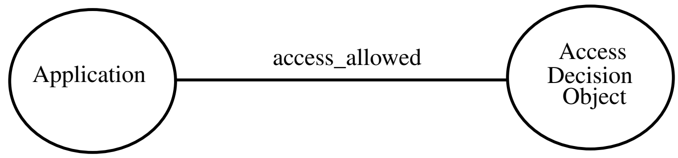


Figure 15-37 access_allowed Application

The application can specify the privileges of the initiating principal and a variety of authorization data, which could include the function being performed, and the data it is being performed on.

An application access policy can be used to supplement the standard invocation access policy with an application-defined policy. Such a policy might, for example, take into account the parameters to the request. In this case, the authorization data passed to the application-defined policy would be likely to include the request's operation, parameters, and target object.

The application access policy could be associated with the domain, and managed using the domain structure as for other policies (see Section 15.4.5.2, "Administrative Model," on page 15-75). In this case, the application obtains the **Access Policy** object as shown in Figure 15-38.

Figure 15-38 get_policy Application

However, the application could choose to manage its access policy differently.

Auditing Application Activities

Applications can enforce their own audit policies, auditing their own activities. Audit policies specify the selection criteria for deciding whether to audit events.

As for application access policies, application audit policies can be associated with domains and managed via the domain structure. No standard application-level audit policy is specified, as different applications may want to use different selectors in deciding which events to audit. Application events are generally not related to object invocations. Applications can provide their own audit policies, which use different criteria. The most common selectors for these audit policies to use are the event type and its success or failure, the **audit_id** and the time. (Management of such policies can generally be done using the interfaces for audit policy administration defined in Section 15.6.5, "Audit Policies," on page 15-144, by specifying new selectors, appropriate to the application concerned.)

Whether or not the application uses an audit policy, it uses an **Audit Channel** object to write the audit records. One Audit Channel object is created at ORB initialization time, and this is used for all system auditing. Applications can use different audit channels.

The way an Audit Channel object handles the audit records is not visible to the caller. It may filter them, route them to appropriate audit trails, or cause event alarms. Different Audit Channel objects may be used to send audit records to different audit trails.

Applications and system components both invoke the **audit_write** operation to send audit records to the audit trail.

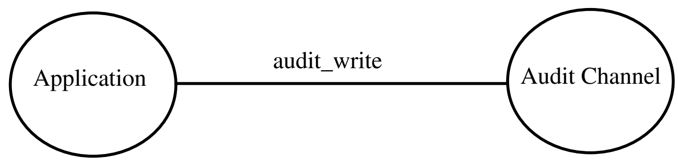


Figure 15-39 audit_write Application

If an application is using an audit policy administered via domains, it uses an **Audit Decision** object (see Section 15.5.7, "Security Audit," on page 15-113) to decide whether to audit an event. It can find the appropriate **Audit Decision** object using the **audit_decision** attribute of **Current** as follows.

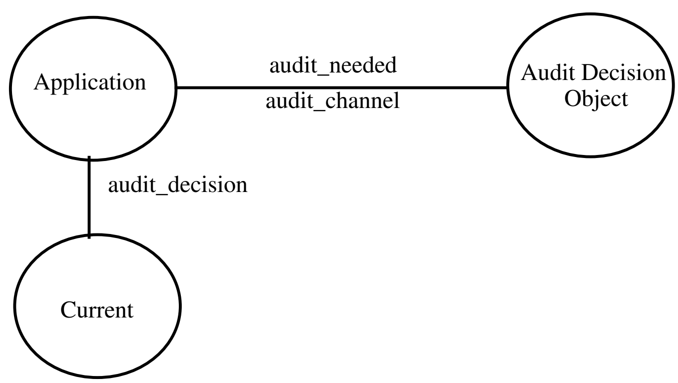


Figure 15-40 Audit Decision Object

The application invokes the **audit_needed** operation of the **Audit Decision** object, passing the values required to decide whether auditing is needed. (This set of selectors could include, for example, the type of event, its success or failure, the identity of the caller, the time, etc. See administration of audit policies in Section 15.5.7, "Security Audit," on page 15-113.)

The audit channel to be used in conjunction with an audit policy object can be identified to the audit policy object with an audit channel id. The **Audit Decision** object uses this **Audit Channel Id** to gain access to the corresponding **Audit Channel** and return it to the user. Thus the application can use an **Audit Channel** associated with the application (and these can link into the system audit services). If so, the application uses the **audit_channel** attribute of the Audit Decision object to find the **Audit Channel** object to use. However, applications can create their own **Audit Channels** with the help of the underlying audit service, and register their **Audit Channel Ids** with the appropriate **Audit Policy** object. The association between the **Audit Channel Id** and the audit channel is maintained by the underlying audit service, which is not specified in this chapter.

Finding What Security Policies Apply

An application may want to find out what policies the system is enforcing on its behalf. For example, it may want to know the default quality of protection to be used by default for messages or for non-repudiation evidence.

To do this, it can call **Current::get_policy**, and then the appropriate **get_** operation of the policy object obtained as defined in Section 15.6, "Administrator's Interfaces," on page 15-130 (if permitted).

Non-repudiation

The non-repudiation services in this specification provide generation of evidence of actions and later verification of this evidence, to prove that the action has occurred. There is often data associated with the action, so the service needs to provide evidence of the data used, as well as the type of action.

These core facilities can be used to build a range of non-repudiation services. It is envisioned that *delivery services* will be implemented to deliver this evidence to where it is needed and *evidence stores* will be built for use by adjudicators. As different services may have different requirements for these, interfaces for them are not included in this specification.

Non-repudiation Credentials and Policies

Non-repudiation operations are performed on **NRCredentials**. As for any other **Credentials** object, these hold the identity and attributes of a principal. However, in this case, the attributes include whatever is needed for identifying the user for generating and checking evidence. For example, it might include the principal's key (or provide access to it) as needed to sign the evidence.

NRCredentials are available via the **Current** object as for other **Credentials** objects, and support the operations defined for credentials previously described. The credentials to be used for non-repudiation can be specified using the **set_credentials** operation on **Current** with a type of **NRCredentials**.

An application can set security attributes related to non-repudiation using the **NRCredentials::set_NR_features** operation (this is similar to the typical usage of **Credentials::set_security_features**).

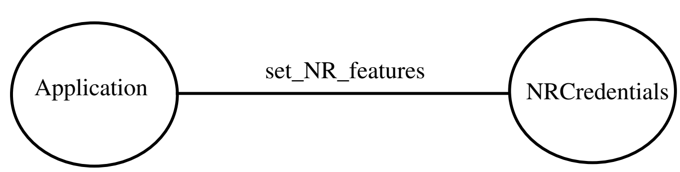


Figure 15-41 set_NR_features Operation

The **set_NR_features** can be used to specify, for example, the quality of protection and the mechanism to be used when generating evidence using these credentials.

By default, the features are those associated with the non-repudiation policy obtained by invoking **Current::get_policy** specifying **Security::SecNonRepudiation**. However, non-repudiation policies may come from other sources. For example, the policy to be used when generating evidence for a particular recipient may be supplied by that recipient.

There is an **NRCredentials::get_NR_features** operation equivalent to **set_NR_features**.

Evidence generation and verification operations are also performed on **NRCredentials** objects. These are described next.

Using Non-Repudiation Services

An application can generate evidence associated with an action so that it cannot be repudiated at a later date. All evidence and related information is carried in non-repudiation tokens. (The details of these are mechanism-specific.)

The application decides that it wishes to generate some proof of an action and calls the **generate_token** operation of an **NRCredentials** object.

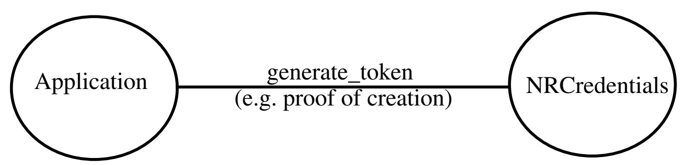


Figure 15-42 generate_token Operation

This evidence is created in the form of a non-repudiation token rendered unforgeable. Generation of the token uses the initiating principal's security attributes in the **NRCredentials** (normally a private key), for example, to sign the evidence.

Depending on the underlying cryptographic techniques used, the evidence is generated as:

- A secure envelope of data based on symmetric cryptographic algorithms requiring what is termed to be a trusted third party as the evidence-generating authority.
- A digital signature of data based on asymmetric cryptographic algorithms which is assured by public key certificates, issued by a Certification Authority.

Depending on the non-repudiation policy in effect for a specific application and the legal environment, additional information (such as certificates or a counter digital signature from a Time Stamping Authority) may be required to complete the non-repudiation information. A time reference is always provided with a non-repudiation token. A Notary service may be required to provide assurance about the properties of the data.

Complete Evidence

Non-repudiation evidence may have to be verified long after it is generated. While the information necessary to verify the evidence (e.g., the public key of the signer of the evidence, the public key of the trusted time service used to countersign the evidence, the details of the policy under which the evidence was generated, etc.) will ordinarily be easily accessible at the time the evidence is generated, that information may be difficult or impossible to assemble a long time afterward.

The CORBA Non-repudiation Service provides facilities for incorporating all information necessary for the verification of a piece of non-repudiation evidence inside the evidence token itself. A token including both non-repudiation evidence and all information necessary to verify that evidence is said to contain "complete" evidence.

There may be policy-related limitations on the time periods during which complete evidence may be formed. For example, Non-repudiation policy may permit addition of the signer's public key to the evidence only after expiration of the interval, during which the signer may permissibly declare that key to have been compromised. Similarly, the policy may require application of the Trusted Time Service countersignature within a specified interval after application of the signer's signature.

To facilitate the generation of complete evidence, the information returned from the calls which verify evidence and request formation of complete evidence, includes two indicators (**complete_evidence_before** and **complete_evidence_after**) indicating the earliest time at which complete evidence may usefully be requested and the latest time at which complete evidence can successfully be formed.

A call to **verify_evidence** before complete evidence can be formed may result in a response declaring the evidence to be "conditionally valid." This means that the evidence is not invalid at the current time, but a future event (e.g., the signer declaring his key compromised) might cause the evidence to be invalid when complete.

Figure 15-43 on page 15-73 illustrates the policy considerations relating to generation of complete evidence, and the sequence of actions involved in generating and using complete evidence.

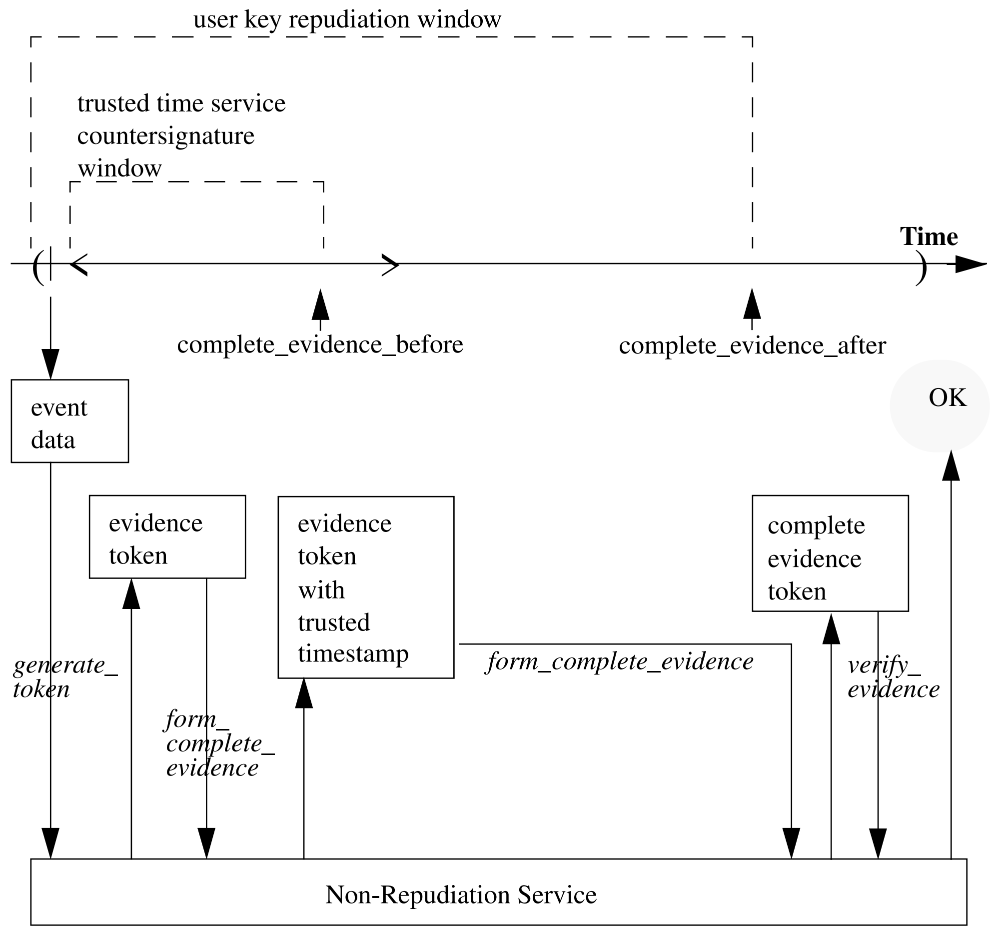


Figure 15-43 Non-repudiation Service

An application may receive a token and need to know what sort of token it is. This is done using **get_token_details**. When the token contains evidence, **get_token_details** can be used to extract details such as the non-repudiation policy, the evidence type, the originator's name, and the date and time of generation. These details can be used to select the appropriate non-repudiation policy and other features (using **set_NR_features**), as necessary for verifying the evidence. When the token contains a request to send back evidence to one or more recipients, then if appropriate, evidence can be generated.

An application verifies the evidence using the **verify_evidence** operation.

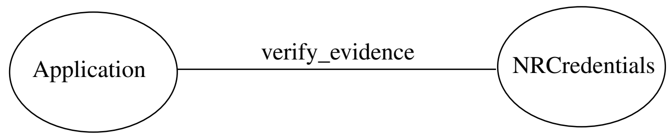


Figure 15-44 verify_evidence operation

Verification of non-repudiation tokens uses information associated with the Non-repudiation Policy applicable to the non-repudiation token and security information about the recipient who is verifying the evidence (normally the public key from a Certification Authority and a set of trust relationships between Certification Authorities).

Using Non-Repudiation for Receipt of Messages

An application receiving a message with proof of origin may handle it as shown in Figure 15-45.

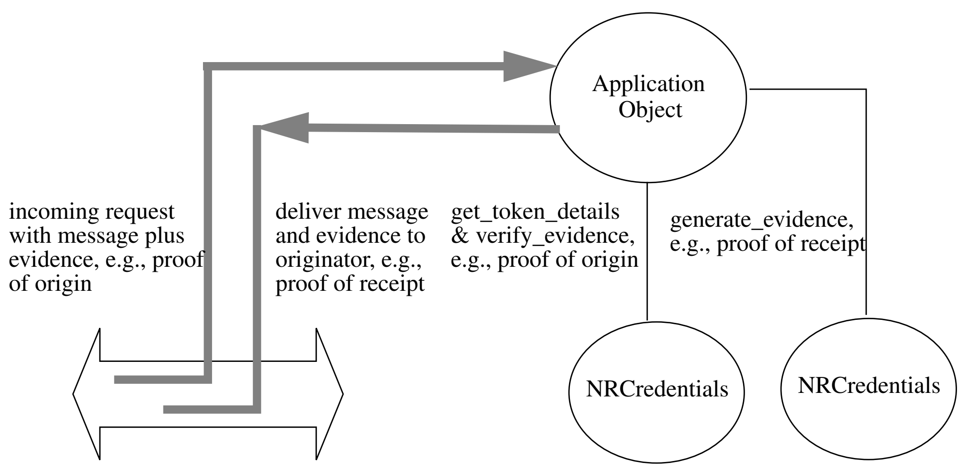


Figure 15-45 Proof of Origin Message

- The application receives the incoming message with a non-repudiation token that has been generated by the originator.
- The application now wishes to know the type of token that it has received. It does this by calling the **NRCredentials::get_token_details** operation. The token may be:
  - A request that evidence be sent back (such as an acknowledgment of receipt)
  - Evidence of an action (such as a proof of creation)
  - Both evidence and a request for further evidence.
- The application's next action depends on which of the three cases applies.
  - In the first case, the application verifies that it is appropriate to generate the requested evidence and, if so, generates that evidence using **NRCredentials::generate_token.**

- In the second case, the application retrieves the data associated with the evidence if it is outside the token, and verifies the evidence using **NRCredentials::verify_evidence,** presenting the token alone or the concatenation of the token and the data.
- In the last case, the application verifies the received evidence by first calling **NRCredentials::verify_evidence,** and then generating evidence if appropriate, as in the first case.

- If the application receives a token that contains valid evidence, and wishes to store it for later use, it needs to make sure that it holds all the necessary information. It may need to call **NRCredentials::form_complete_evidence** in order to get the complete evidence needed when this could not be provided using the verify operation.
- When the application has generated evidence as the result of a request from the originator of the message, the application must send it to the various recipients as indicated in the NR token received.

Using Non-repudiation Services for Adjudication

Adjudication applications use the **NRCredentials::verify_evidence** operation, which must return complete evidence to settle disputes.

15.4.5.2 Administrative Model

The administrative model described here is concerned with administering security policies.

- Administration of security environment domains and security technology domains may be implementation-specific, so it is not covered here. This means administrating security-technology-specific objects is out of the scope of this specification.
- Explicit management of nonsecurity aspects of domains is not covered.

Administrative activities covered here are:

- Creating objects in a secure environment subject to the security policies
- Finding the domain managers that apply to this object.
- Finding the policies for which these domain managers are responsible.
- Setting security policy details for these policy objects.
- Specifying which rights give access to which operations in support of access policies.

The model used here is not specific to security, though the specific policies described are security policies.

Security Policies

Security policies may affect the security enforced:

- By applications. In general, enforcing policy within applications is an application concern, so it is not covered by this specification. However, where the application uses underlying security services, it will be subject to their policies.
- By the ORB Security Services during object invocation (the main focus of this specification).
- In other security object services, particularly authentication and audit.
- In any underlying security services. (In general, this is not covered by this specification, as these security services are often security-technology-specific.)

This specification defines the following security policy types:

- ***Invocation access policy* -** The object that implements the access control policy for invocations of objects in this domain.
- ***Invocation audit policy* -** This controls which types of events during object invocation are audited, and the criteria controlling auditing of these events.
- ***Secure invocation policy* -** This specifies security policies associated with security associations and message protection. For example, it specifies:
  - Whether mutual trust between client and target is needed (i.e., mutual authentication if the communications path between them is not trusted).
  - Quality of protection of messages (integrity and confidentiality).

There may be separate invocation policies for applications acting as client and those acting as target objects in this domain. This applies to access, audit, and secure invocation policies. There may also be separate policies for different types of objects in the domain.

- ***Invocation delegation policy* -** This controls whether objects of the specified type in this domain, when acting as an intermediate in a chain, by default delegate the received credentials, use their own credentials, or pass both.
- ***Application access policy* -** This policy type can be used by applications to control whether application functions are permitted. Unlike invocation policies, it does not have to be managed via the domain structure, but may be managed by the application itself.
- ***Application audit policy* -** This policy type can be used by applications to control which types of application events should be audited under what circumstances.
- ***Non-repudiation policy* -** Where non-repudiation is supported, a non-repudiation policy has the rules for generation and verification of evidence.
- ***Construction policy* -** This controls whether a new domain is created when an object of a specific type is created.

Domains at Object Creation

Any object that is accessible through an ORB must have an object reference created for it. This is often done as a part of the procedure for creating the object by a factory object. When a new object reference is created in a secure environment, the ORB implicitly associates the object reference, and hence the associated object, with the following elements forming its environment.

- One or more *Security Policy Domains*, defining all the policies to which the object is subject.
- The *Security Technology Domains,* characterizing the particular variants of security mechanisms available in the ORB.
- Particular *Security Environment Domains* where relevant.

The application code involved in the creation of an object, and its reference may not need to be aware of security to protect the objects it creates, if the details are encapsulated in a Factory object. Automatically making an object reference and hence the associated object a member of policy domains on creation ensures that mandatory controls of enclosing domains are not bypassed.

The ORB will establish these associations when the creator of the object calls **PortableServer::POA::create reference** or **PortableServer::POA::create_reference_with_id** (see the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*) or an equivalent. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, a new domain is also needed. For example, in a banking system, there may be a domain for each bank branch, which provides policies for bank accounts at that branch. Therefore when a bank branch is created, a new domain is needed. As for a newly created object's domain membership, if the application code creating the object and the object reference to it is to be unaware of security, the domain manager must be created transparently to the application. A construction policy specifies whether new objects reference of this type in this domain require a new domain.

This construction policy is enforced at the same time as the domain membership (i.e., by **POA::create_reference*** or equivalent). For details, see the Portable Object Adaptor chapter of the *Common Object Request Broker: Architecture and Specification*.

Other Domain and Policy Administration

Once an object reference has been created as a member of a policy domain, it may be moved to other domains using the appropriate domain management facilities (not specified in this chapter).

Once a domain manager has been created, new security policy objects can be associated with it using the appropriate domain management facilities. These security policy objects are administered as defined in this specification.

The following diagram shows the operations needed by an administrative application to manage security policies.

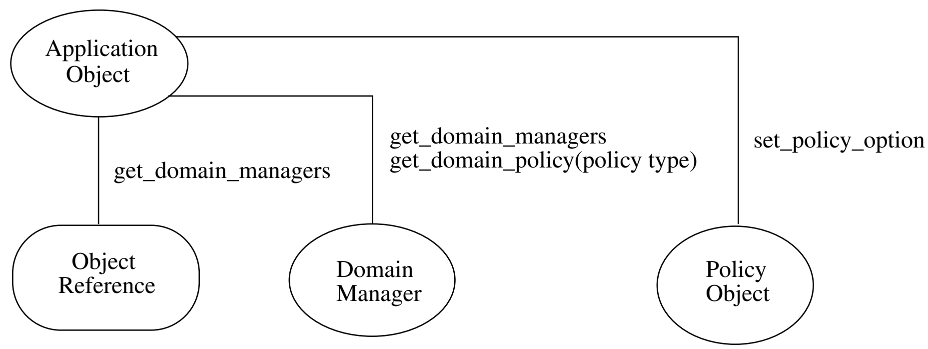


Figure 15-46 Managing Security Policies

Finding Domain Managers

An application can invoke the **get_domain_managers** operation on an object reference to obtain a list of the immediately enclosing domain managers for that object (i.e., the object associated with the object reference). If these do not have the type of policy required, a call can be made to **get_domain_managers** on one of these domain managers to find its immediately enclosing domains.

Finding the Policies

Having found a domain manager, the administrative application can now find the security policies associated with that domain by calling **get_domain_policy** on the domain manager specifying the type of policy it wants (e.g., client-secure invocation policy, application audit policy). This returns the **Policy** object needed to administer the policy associated with this domain. Each **Policy** object supports the operations required to administer that policy.

In this specification, no facilities are provided to specify the rules for combining policies for overlapping domains, though some implementations may include default rules for this. (Definition of such rules is a potential candidate for future security specifications. See Appendix F, "Facilities Not in This Specification" on page 15-358.)

If the policy that applies to the domain manager's own interface is required (rather than the one for the objects in the domain), then **get_policy** (rather than **get_domain_policy**) is used.

Setting Security Policy Details

Having found the required security Policy object, the application uses its interface to set the policy.

The operations available through the interface depend on the type of policy. For example, the delegation policy only requires a delegation mode to be set to specify delegation mode used when the object acts as an intermediate in a chain of object invocations, whereas an access policy will need to have an operation that makes it possible to specify who can access the objects.

Administrative interfaces are defined in Section 15.6, “Administrator’s Interfaces,” on page 15-130, for the standard policy types, which all ORBs supporting security functionality Level 2 support.

Different administration may be needed if standard policies are replaced by different policies. A supplier providing another policy may therefore have to specify its administrative interfaces.

Specifying Use of Rights for Operation Access

The access policy is used to decide whether a user with specified privileges has specified *rights*. A specific right may permit access to exactly one operation. More often, the right permits access to a set of operations.

A **RequiredRights** object specifies which rights are required to use which operations of an interface. The administrator can **set_required_rights** on this object.

15.4.5.3 The Model as Seen by the Objects Implementing Security

Security is provided for security-unaware applications by implementation-level security objects, which are not directly accessible to applications. These same implementation objects are also used to support the application-visible security objects and interfaces described in “The Model as Seen by Applications” on page 15-58 and “Administrative Model” on page 15-75.

There are two places where security is provided for applications, which are unaware of security. These are:

1. On object invocation when invocation time policies are automatically enforced.
2. On object creation, when an object automatically becomes a member of a domain, and therefore subject to the domain’s policies.

Implementor’s View of Secure Invocations

Figure 15-47 on page 15-80 shows the implementation objects and services used to support secure invocations.

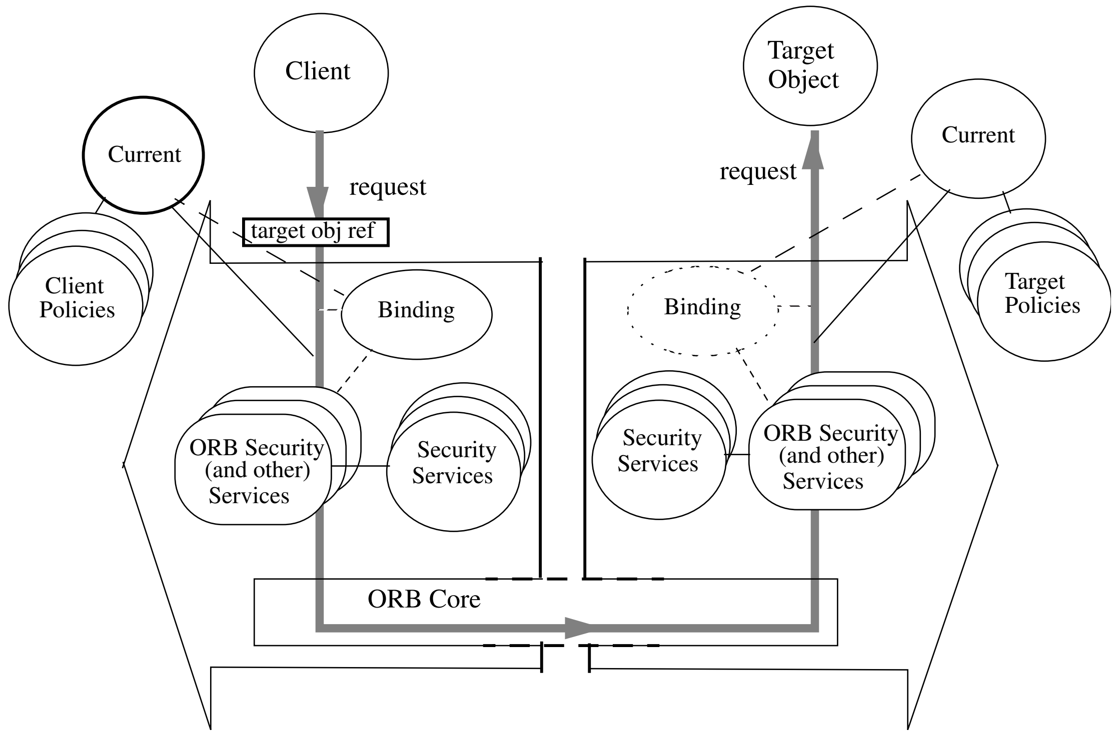


Figure 15-47 Securing Invocations

ORB Security Services

ORB Security Services are interposed in the path between the client and target object to handle the security of the object invocation. They may be interspersed with other ORB services, though where message protection is used, this will be the last ORB service at the client side, as the request cannot be changed after this.

The ORB services use the policy objects to find which policies to apply to the client and target object, and hence the invocation. The ORB and ORB Services establish the binding between client and target object as defined in ORB Services, under Section 15.4.2, "Structural Model," on page 15-49. The ORB Security Services call on the security services to provide the required security.

Security Policy

At the client, the security policies associated with it are accessed by the ORB Security Services using the **Current::get_policy** operation specifying the type of policy required. At the client, the invocation policies that will be used for a specific invocation through a specific object reference can be inspected using the **get_policy** operation on that object reference. At the target, **Current::get_policy** is used in a similar way to obtain the policy associated with the target object.

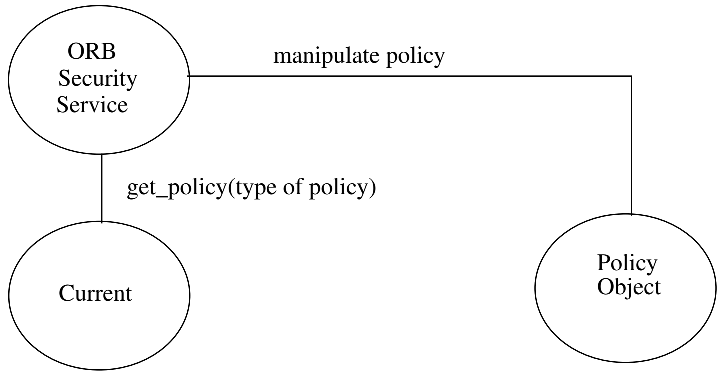


Figure 15-48 get_policy Operation

Once the policy object has been obtained, the ORB Service uses it to enforce policy. The operations used to enforce the policy depend on the type of policy. In some cases, such as secure invocation or delegation, the ORB Service invokes a **get_** operation of the appropriate **Policy** object (e.g., **SecureInvocationPolicy::get_association _options, DelegationPolicy::get_delegation_mode**) specifying the particular policy options required (e.g., whether confidentiality is required, and the delegation mode, respectively). It then uses this information to enforce the policy, for example, pass the required policy options to the **Vault** to enforce.

Decision objects corresponding to certain policy objects include rules, which enforce the policy. For example, an access decision object corresponding to the access policy object has the **access_allowed** operation which responds with a yes or no.

Specific ORB Security Services and Replaceable Security Services

The specific ORB Security Services and security services included in the CORBA security object model are shown in Figure 15-49 on page 15-82.

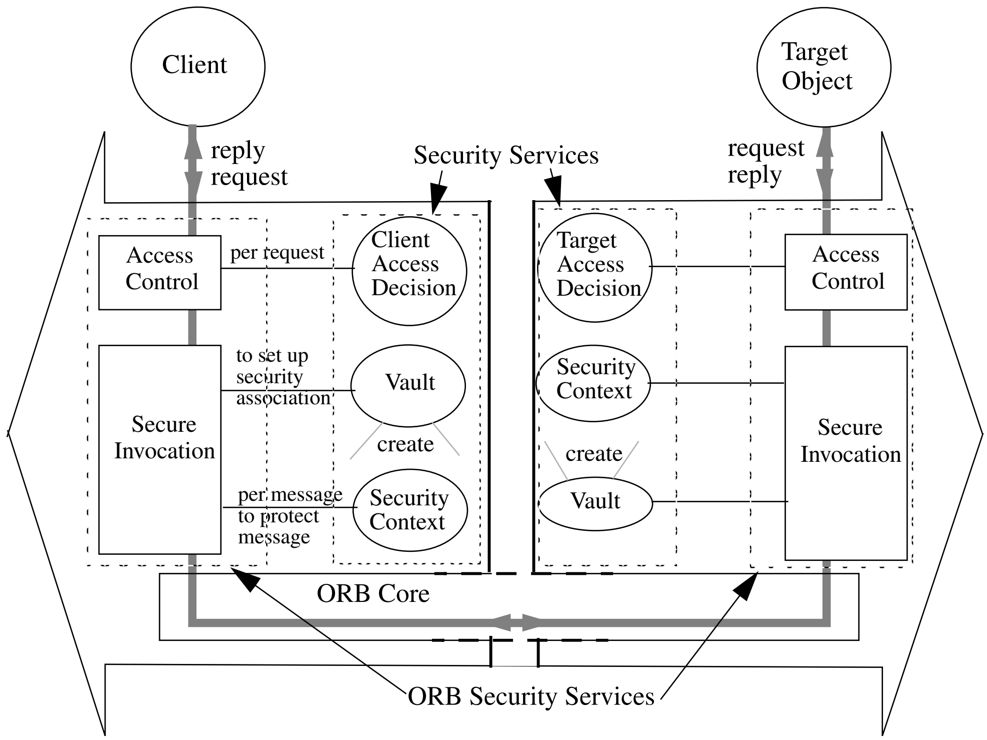


Figure 15-49 ORB Security Services

Two ORB Security Services are shown:

1. The access control service, which is responsible for checking if this operation is permitted and enforcing the invocation audit policy for some event types.
2. The secure invocation service. On the client's initial use of this object, it may need to establish a security association between client and target object. It also protects the application requests and replies between client and target object.

The security services they use are discussed next.

Access Policy

An **Access Decision** object is used to determine if a given operation on a specific target object is permitted. It is obtained by the ORB service using the **access_decision** attribute of the **Current** object. Since the **Access Decision** objects are **locality-constrained**, of necessity the access decision objects at the client and target are distinct.

The ORB service invokes the **access_allowed** operation on the **Access Decision** object specifying the operation required, the principal credentials to be used for deciding if this access is allowed, etc. This is independent of the type of access control policy, which may be discretionary using ACLs or capabilities, mandatory labels usage, etc.

The **Access Decision** object uses the access policy to decide what rights the principal has by invoking the **get_effective_rights** operations on the appropriate **Access Policy** object.

If the access policies use **rights** (rather than directly identifying that this operation is permitted), the **Access Decision** object now invokes **get_required_rights** on the **RequiredRights** object to find what rights are needed for this operation. It compares these rights with the effective rights granted by the policy objects, and if required rights have been granted, it grants access. This model could be extended in the future to handle overlapping access policy domains as described in Appendix F, "Facilities Not in This Specification" on page 15-358.

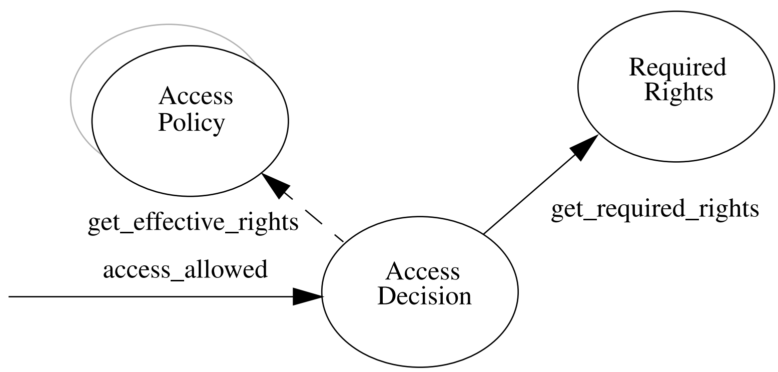


Figure 15-50 Access Decision Object

Vault

The **Vault** object is responsible for establishing the security association between client and target. It is invoked by the Secure Invocation ORB Service at the client and at the target (using **init_security_context** and **accept_security_context**). The **Vault** creates the security context objects, which are used for any further security operations for this association.

Authentication of users (and some other principals) is done explicitly using the **authenticate** operation described in Section 15.5.3, "Authentication of Principals," on page 15-90. Authentication of an intermediate object in a chain (or the principal representing the object) may be done automatically by the **Vault** when an intermediate object invokes another object.

The **Vault**, like the security context objects it creates, is invisible to all applications.

Security Context

For each security association, a pair of **Security Context** objects (one associated with the client, and one with the target) provide the security context information. Establishing the security contexts may require several exchanges of messages containing security information, for example, to handle mutual authentication or negotiation of security mechanisms.

Security Context objects maintain the state of the association, such as the credentials used, the target's security name, and the session key. The **is_valid** and **refresh** operations are supported to check the validity of the context and refresh it if possible.

Security Context objects provide operations for protecting messages for integrity and confidentiality such as **protect_message** and **reclaim_message**.

They also have **received_credentials** and (received) **security_features** attributes, which are made available via the **Current** object.

A security context can persist for many interactions and may be shared when a client invokes several target objects in the same trusted identity domain. Although neither the client nor target is aware of an "association," it is an important optimizing concept for the efficient provision of security services.

Relationship between Implementation Objects for Associations

There is not always a one-for-one relationship between client-target object pairs and security contexts. For example, if a client uses different privileges for different invocations on that object, this will result in separate security contexts. Also, a security context may be shared between this client's calls on more than one target object. This is normally the case if the target objects share a security name, as shown in Figure 15-51 on page 15-85. Note that the **Vault** decides whether to use the same or a different security context based on the target security name (which may be the name of an object or trusted identity domain).

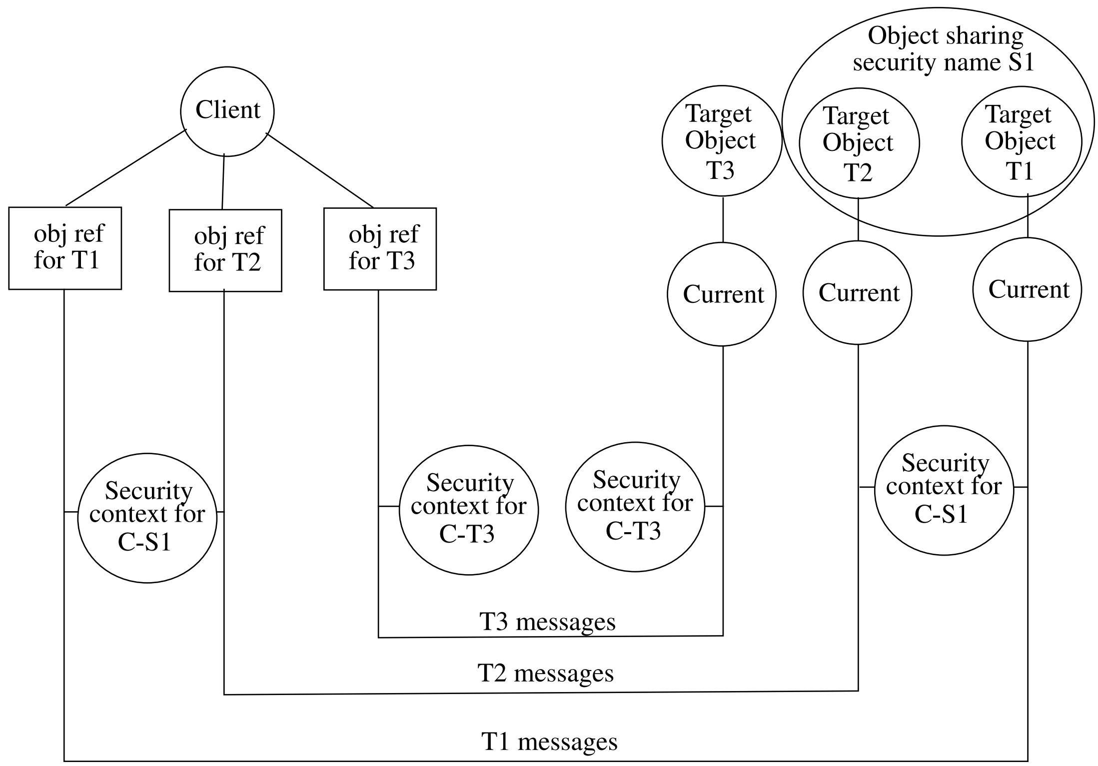


Figure 15-51 Target Objects Sharing Security Names

Implementor's View of Secure Object Creation

When an object is created in a secure environment, it is associated with Security Policy, Environment, and Technology domains as described in "Administrative Model" on page 15-75.

The way it is associated with Environment and Technology domains is ORB implementation-specific, and therefore not described here.

For policy domains, the construction policy of the application or factory creating the object is used as shown in Figure 15-52 on page 15-86.

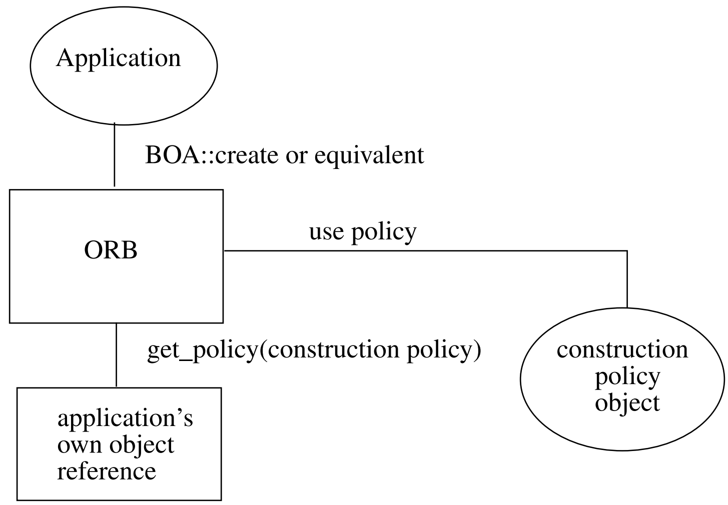


Figure 15-52 Object Created by Application or Factory

The application (which may be a generic factory) object calls **POA::create_reference** or equivalent to create the new object reference. The ORB obtains the construction policy associated with the object reference to be created. If the application that is attempting to create the object reference is itself a CORBA object, then the ORB attempts to obtain the construction policy associated with it. If the ORB is unable to obtain a construction policy for the object reference to be created, it uses a default construction policy, which does not create a new domain.

The construction policy controls whether, in addition to creating the specified new object reference, the ORB must also create a new domain. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object.

If a new domain is not needed and the application is itself not an object and hence has no domain associated with it, the ORB uses a default domain to place the newly created object reference. In all cases a reference to the domain manager associated with the newly created object reference can be obtained by calling **get_domain_managers** on the newly created object's reference (See the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*).

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain, or an ORB-specific default set of policies in the case that the object reference was created in a situation where there is no enclosing domain (e.g., by an application that is itself not a CORBA object and hence has no domain associated with it).

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management operations. Please note: these operations do not form a part of this specification.

15.4.5.4 *Summary of Objects in the Model*

The previous sections have described the various security-related objects, which are available to applications, administrators, and implementors.

Figure 15-53 shows the relationship between the main objects visible in different views for three types of security functionality.

1. Authentication of principals and security associations (which includes authentication between clients and targets) and message protection.
2. Authorization and access control (i.e., the principal being authorized to have privileges or capabilities and control of access to objects).
3. Accountability -- auditing of security-related events and using non-repudiation to generate and check evidence of actions.

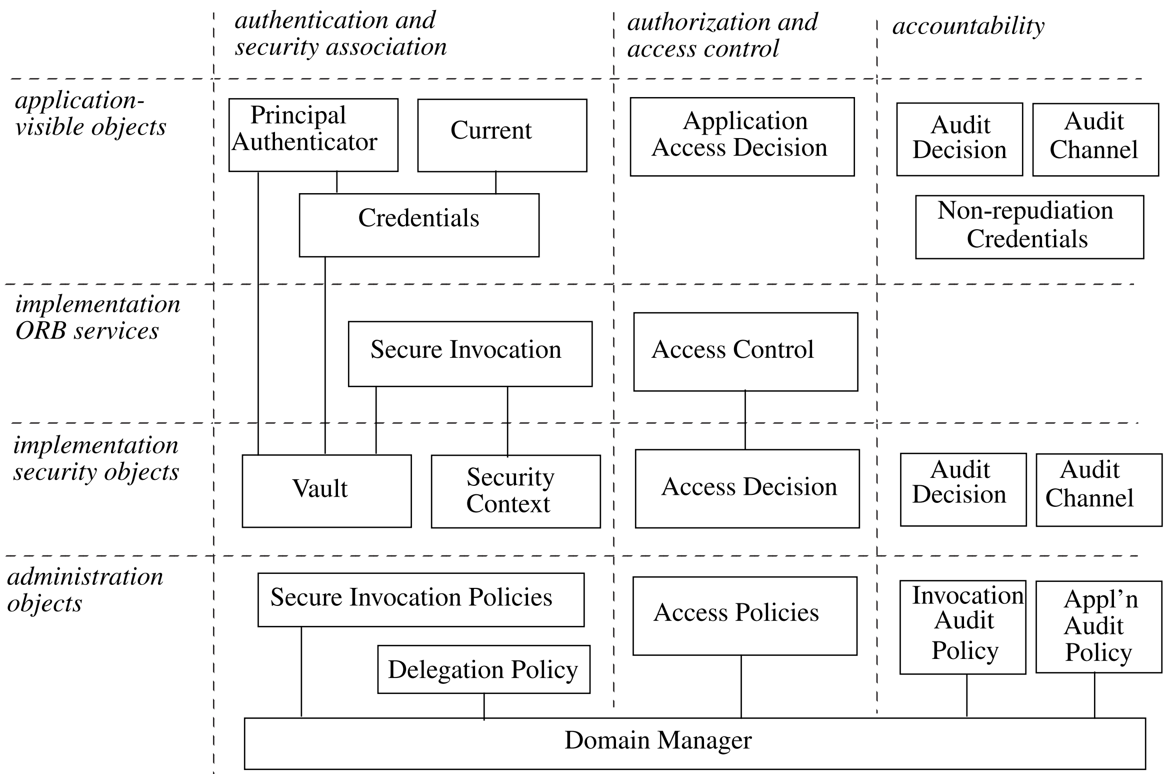


Figure 15-53 Relationship Between Main Objects

Credentials are visible to the application after authentication, for setting or obtaining privileges and capabilities, for access control, and are available to ORB service implementors. Only the first of these usages is shown.

Policy objects have management operations to allow policies to be maintained. These operations depend on the type of policy. For example, management of a mandatory access control policy using labels is different from management of an ACL. However, at run-time, an access decision object is used, which has a standard "check if access is allowed" operation, whatever the access control policy used. The access policy object has the management operations, whereas the access decision object has the runtime decision operations.

The diagram does not show:

- Application objects (client, target object, target object reference at the client).
- The ORB core (though the security ORB services it calls are shown).
- The construction policy object.

15.5 Application Developer's Interfaces

15.5.1 Introduction

This section defines the security interfaces used by the application developer who implements the business logic of the application. For an overview of how these interfaces are used, see "Application Developer's View" on page 15-47.

Please note that applications may be completely unaware of security, and therefore not need to use any of these interfaces. In general, applications may have different levels of security awareness. For example:

- Applications unaware of security, so that an application object, which has not been designed with security in mind, can participate in a secure object system and be subject to its controls such as:
- Protection default quality of protection on object invocations.
  - Control of who can perform which operations on which objects.
  - Auditing of object invocations.
- Applications performing security-relevant activities. An application may control access and audit its functions and data at a finer granularity than at object invocation.
- Applications wanting some control of the security of its requests on other objects, for example, the level of integrity protection of the request in transit.
- Applications that are more sophisticated in how they want to control their distributed operations, for example, control whether their credentials can be delegated.
- Applications using more specialist security facilities such as non-repudiation.

Security operations use the standard CORBA exceptions. For example, any invocation that fails because the security infrastructure does not permit it, will raise the standard CORBA::NO_PERMISSION exception. A security operation that fails because the feature requested is not supported in this implementation will raise a CORBA::NO_IMPLEMENT exception. Any parameter that has inappropriate values should be flagged by raising the CORBA::BAD_PARAM exception. No security-specific exceptions are specified.

15.5.1.1 Security Functionality Packages

Two security functionality packages and one optional security functionality package are defined in this specification. In addition, the Security-Ready functionality packages are also described in this and the two following sections.

Security Functionality Level 1 Package

Security functionality Package 1 provides an entry level of security functionality that applies to all applications running under a secure ORB, whether aware of security or not. This includes security of invocations between client and target object, message protection, some delegation, access control, and audit.

Generally, the security functionality is specified by administering the security policies for the objects, and is mainly transparent to applications.

Security Functionality Level 1 Package includes operations for applications as follows: **Current::get_attributes** allows an application to obtain the privileges and other attributes of the principal on whose behalf it is operating. It can then use these to control access to its own functions and data (see Section 15.5.4, "The Credentials Object," on page 15-94, and Section 15.5.9, "Access Control," on page 15-117).

Security Functionality Level 2 Package

This security functionality level provides further security functionality such as more delegation options.

It also allows an application aware of security to have more control of the enforcement of this security. Most of the interfaces specified in this section are only available as part of this functionality level. Note that although implementations must support all Level 2 interfaces in order to conform to Security Functionality Level 2, different implementations of these interfaces may support different semantic extensions, while maintaining the same core semantics; some implementations will therefore be capable of enforcing a wider variety of policies than others.

Optional Functionality Package

The only specified optional facility specified here is non-repudiation. The interfaces for this are specified in Section 15.5.11, "Non-repudiation," on page 15-121.

It is possible to add other security policies to this specification, for example, extra access or delegation policies, but these are not part of this specification.

15.5.1.2 Introduction to the Interfaces

The interfaces specified here, as in other sections, are designed to allow a choice of security policies and mechanisms. Where possible, they are based on international standard interfaces. Several of the operations in the **Credentials** interface are based on those of GSS-API.

Data Types

Many of the security data types used by applications are also used for implementation interfaces; therefore, these are defined in a separate module called **Security.** See Appendix Section A.10, "General Security Data Module," on page 15-283 for the details of the data types used by the interfaces.

Some data types, such as security attributes and audit events, have an extensible set of values, so the user can add values as required to meet user-specific security policies. In these cases, a family is identified, and then a set of types or values for this family. Family identifiers 0-7 are reserved for OMG-defined families, and therefore standard values. More details of these families and associated data types are given in Appendix Section A.19, "Values for Standard Data Types" on page 15-307.

In the interface specifications in the rest of this section, data types defined in module **Security** are included without the qualifying **Security::** for ease of readability. The full definitions are included in Appendices A and B.

15.5.2 Finding Security Features

15.5.2.1 Description of Facilities

An application can find out what security facilities this implementation supports, for example, which security functionality level and options it supports. It can also find out what security technology is used to provide this implementation.

The **CORBA::ORB::get_service_information** operation is used to determine what security features are supported by this ORB (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*). To request information about Security service the **CORBA::ServiceType** constant value, **CORBA::Security** should be used. Refer to the constant definitions of type **CORBA::SecurityOptions** in the IDL Security module in Appendix Section A.10, "General Security Data Module" on page 15-283 to see the definition of various service options relevant to security.

15.5.3 Authentication of Principals

15.5.3.1 Description of Facilities

A principal must establish its credentials before it can invoke an object securely. For many clients, there are default credentials, created when the user logs on. This may be performed prior to using any object system client. These default credentials are automatically used on object invocation without the client having to take specific action. Even if user authentication is executed within the object system, it should normally be done by a user sponsor/login client, which is separate from the business application client, so that business applications can remain unaware of security.

In most cases, principals must be authenticated to establish their credentials. However, some services accept requests from unauthenticated users. In this case, if the principal has no credentials at the time the request is made, unauthenticated credentials are created automatically for it.

If the user (or other principal) requires authentication and has not been authenticated prior to calling the object system, the (login) client must invoke the **Principal Authenticator** object to authenticate, and optionally select attributes for, the principal for this session. This creates the required **Credentials** object and makes it available as the default credentials for this client. Its object reference is also returned so it can be used for other operations on the **Credentials**. If the object system supports non-repudiation, the credentials returned can be used for non-repudiation operations as specified in "Non-repudiation" on page 15-121.

Authentication of principals may require more than one step, for example, when a challenge/response or other multistep authentication method is used. In this case, the authentication service will return information to the caller, which may be used in further interactions with the user before continuing the authentication. So there are both **authenticate** and **continue_authentication** operations of the **Principal Authenticator** object.

There is no need for an application to explicitly authenticate itself to act as an initiating principal prior to invoking other objects, as this will be performed automatically if needed. However, it does need to be performed explicitly if the object wants to specify particular attributes.

The Principal Authenticator object is a **locality-constrained** object.

15.5.3.2 The SecurityLevel2::PrincipalAuthenticator Interface

This section describes the **PrincipalAuthenticator** interface that has the following operations.

authenticate

This is called, for example, by a user sponsor to authenticate the principal and optionally request privilege attributes that the principal requires during its session with the system. It creates a **Credentials** object including the required attributes.

```
AuthenticationStatus authenticate(
    in      AuthenticationMethod    method,
    in      SecurityName            security_name,
    in      Opaque                  auth_data,
    in      AttributeList           privileges,
    out     Credentials             creds,
    out     Opaque                  continuation_data,
    out     Opaque                  auth_specific_data
);
```

Parameters

method	The identifier of the authentication method used
security_name	The principal's identification information (e.g., login name).
auth_data	The principal's authentication information such as password or long term key.
privileges	The privilege attributes requested.
creds	Object reference of the newly created **Credentials** object. Not fully initialized, therefore, unusable unless return parameter is '**SecAuthSuccess**.'
auth_specific_data	Information specific to the particular authentication service used
continuation_data	If the return parameter from the authenticate operation is '**SecAuthContinue**,' then this parameter contains challenge information for authentication continuation.

Return Value

The return parameter is used to specify the result of the operation.

'SecAuthSuccess'	Indicates that the object reference of the newly created initialized credentials object is available in the **creds** parameter.
'SecAuthFailure'	Indicates that authentication was in some way inconsistent or erroneous, and therefore credentials have not been created.
'SecAuthContinue'	Indicates that the authentication procedure uses a challenge/response mechanism. The **creds** contains the object reference of a partially initialized **Credentials** object. The **continuation_data** indicates details of the challenge.
'SecAuthExpired'	Indicates that the authentication data contained some information, the validity of which had expired (e.g., expired password). **Credentials** have therefore not been created.

continue_authentication

This continues the authentication process for authentication procedures that cannot complete in a single operation. An example of this might be a challenge/response type of authentication procedure.

```
AuthenticationStatus continue_authentication(
    in      Opaque                  response_data,
```

```
    in      Credentials             creds,
    out     Opaque                  continuation_data,
    out     Opaque                  auth_specific_data
);
```

Parameters

response_data	The response data to the challenge.
creds	Reference of the partially initialized Credentials object. The Credentials object is fully initialized only when return parameter is '**SecAuthSuccess**.'
continuation_data	If the return parameter from the continue_authentication operation is '**SecAuthContinue**,' then this parameter contains challenge information for authentication continuation.
auth_specific_data	Information specific to the particular authentication service used.

Return Value

The return parameter is used to specify the result of the operation.

'SecAuthSuccess'	Indicates that the **Credentials** object whose reference was identified by the **creds** parameter is now fully initialized.
'SecAuthFailure'	Indicates that the response data was in some way inconsistent or erroneous, and that therefore **credentials** have not been created.
'SecAuthContinue'	Indicates that the authentication procedure requires a further challenge/response. The **Credentials** object whose reference was identified in the **creds** parameter is still only partially initialized. The **continuation_data** indicates details of the next challenge.
'SecAuthExpired'	Indicates that the authentication data contained some information whose validity had expired (e.g., expired password). The **Credentials** object referred to by the **creds** parameter is not valid.

15.5.3.3 Portability Implications

The **authenticate** and **continue_authentication** operations allow different authentication methods to be used. However, methods available are dependent on availability of underlying authentication mechanisms. This specification does not dictate that particular mechanisms should be used. However, use of some mechanisms, (e.g., those involving hardware such as smart cards or finger print readers) may also require use of device-specific objects so the client using such objects will not be portable to systems which do not support such devices. It is therefore recommended that use of both the **authenticate** operations described here and any device-specific ones be confined to a user sponsor or login client, or that such authentication is done prior to calling the object system, where the credentials resulting from this can be used in portable applications.

15.5.4 The Credentials Object

15.5.4.1 Description of Facilities

A **Credentials** object represents a principal's credential information for the session and therefore includes information such as that principal's privilege attributes and identities such as the audit id. (It also includes some security-sensitive data required when this principal is involved in peer entity authentication. However, such data is not visible to applications.)

Credentials object is a **locality-constrained** object.

An application may want to:

- Specify security invocation options to be used by default whenever these credentials are used for object invocations.
- Modify the privilege and other attributes in the credentials, for example, specify a new role or a capability. This can modify the current privileges in use, or the application can make a copy of the **Credentials** object first, and then modify the new copy.
- Inquire about the security attributes currently in the credentials, particularly the privilege attributes.
- Check if the credentials are still valid or if they have timed out, and if so, refresh them.

Credentials objects are created as the result of:

- Authentication (see "Authentication of Principals" on page 15-90).
- Copying an existing **Credentials** object.
- Asking for a **Credentials** object via **Current** (see Section 15.5.6, "Security Operations on Current," on page 15-105).

The way these credentials are made available for use in invocations is described in Section 15.4, "Security Architecture," on page 15-45, and defined in detail in Section 15.5.5, "Operations on Object Reference," on page 15-99, and Section 15.5.6, "Security Operations on Current," on page 15-105.

Credentials used for non-repudiation also support further facilities as described in Section 15.5.11, "Non-repudiation," on page 15-121.

15.5.4.2 The SecurityLevel2::Credentials Interface

The following operations are in the **Credential** interface.

copy

This operation creates a new **Credentials** object, which is an exact duplicate (a "deep copy") of the **Credentials** object which is the target of the invocation. The return value is a reference to the newly created copy of the original **Credentials** object.

Credentials copy();

Parameters

None.

Return Value

An object reference to a copy of the Credentials object, which was the target of the call.

destroy

This operation destroys the **Credentials** object that it is invoked on. In general, the caller is always responsible for destroying its copy of the **Credentials** object after it is done with it. When **Credentials** are used as **"in"** parameters the callee always makes a copy if needed. Then onwards the callee is responsible for managing the life-style of the copy that it makes. In case of **Credentials** objects that are returned as a result, the caller is responsible for destroying it. In case of **"out"** parameters, the callee is responsible for creating it and the caller is responsible for destroying it. The caller is responsible for providing thread safety for **Credentials** parameters that are passed as **"in"** parameters. They must ensure that no other thread modifies the object until the invoked operation is completed.

void destroy();

Parameters

None

Results

None. The Credentials object is destroyed.

set_security_features

This associates a set of security features with a **Credentials** object and sets each feature to be "on" or "off." The security features affect how a secure association is set up, such as what delegation mode to use, whether trust in the target is needed, and what message protection is required.

Some implementations may allow the security features to be set for communication in one direction only (i.e., for requests only, or replies only) via the direction parameter, but this support is not required for compliant implementations. The request-only and reply-only feature sets are treated as overrides to the "both" feature set. If an unsupported direction is passed to **set_security_features**, the CORBA::BAD_PARAM exception should be raised.

The value of a security feature set by this operation is used for invocations using this **Credentials** object (if this does not contravene the **SecClientSecureInvocation** policy for that feature or the target's invocation policy). Once associated with the **Credentials** object, a feature may be turned "on" or "off" again with an additional call to **set_security_feature**.

```
void set_security_features(
    in      CommunicationDirection          direction,
    in      SecurityFeatureValueList        security_features
);
```

Parameters

direction	The communication direction (i.e., both, request, or reply) to which the security feature should be applied. Normally set to both.
security_features	A sequence of required feature-value pairs. They may indicate the delegation mode or a secure association option such as a message protection requirement, or whether trust in the target is needed. To set the feature on, set the boolean value to TRUE; a value of FALSE is used to turn off the feature.

get_security_features

Returns the security features associated with the **Credentials**.

The **direction** parameter indicates which set of security features (i.e., those set for the request direction, the reply direction, or both) should be returned. Conforming implementations are not required to support the "**request**" and "**reply**" directions. If an unsupported direction is passed to **get_security_features**, the CORBA::BAD_PARAM exception should be raised.

```
SecurityFeatureValueList get_security_features(
    in      CommunicationDirection              direction
);
```

Parameters

direction	The communication direction (i.e., both, request, or reply) to which the security feature should be applied. Normally set to both.

Return Value

A sequence of required feature-value pairs. A boolean value of TRUE indicates the feature is on; a value of FALSE indicates the feature is off.

set_privileges

Used to request a set of privilege attributes (such as role, groups), updating the state of the supplied **Credentials** object. One of the attributes requested may be an attribute set reference, which causes a set of attributes to be requested.

Note – This operation can only be used to set privilege attributes. Other attributes, such as the audit identity, are generated by the system and cannot be changed by the application.

```
boolean set_privileges(
    in      boolean           force_commit,
    in      AttributeList     requested_privileges,
    out     AttributeList     actual_privileges
);
```

Parameters

force_commit	If true, the attributes should be applied immediately; otherwise, attribute acquisition may be deferred to when required by the system.
requested_privileges	A set of (typed) privilege attribute values. One of these may be a role name, which is an attribute set reference used to select a set of attributes. (A null attribute set requests default attributes.) Attributes can include capabilities.
actual_privileges	The set of (typed) privileges actually obtained.

Return Value

TRUE	Indicates that attributes can be set, and that the **actual_privileges** parameter contains the complete set or subset of those attributes requested. It is the responsibility of the application programmer to interrogate the returned attributes to determine their suitability.
FALSE	Operation failed, **Credentials** were not modified.

get_attributes

Used to get privilege and other attributes from the **Credentials**. It can be used to:

- Get privilege attributes, including capabilities, for use in access control decisions. If the principal was not authenticated, only one privilege attribute is returned. This has type *Public* and no meaningful value.
- Get other attributes such as audit or charging identities if available. (If the principal is not authenticated, none of these are returned.)

```
AttributeList get_attributes(
    in      AttributeTypeList          attributes
);
```

Parameters

attributes	The set of security attributes (privilege attributes and identities) whose values are desired. If this list is empty, all attributes are returned.

Return Value

The requested set of attributes reflecting the state of the **Credentials**.

is_valid

Credentials objects may have limited lifetimes. This operation is used to check if the **Credentials** are still valid.

```
boolean is_valid(
    out         UtcT    expiry_time
);
```

Parameters

expiry_time	The time that the Credentials expire.

Return Value

TRUE	The Credentials is still valid.
FALSE	The Credentials is not valid anymore.

refresh

Allows the application to update expired **Credentials**.

boolean refresh ();

Parameters

None

Return Value

TRUE	The **Credentials** were refreshed successfully.
FALSE	The **Credentials** could not be refreshed.

15.5.4.3 Portability Implications

The **PrincipalAuthenticator::authenticate** and **Credentials::set_privileges** operations allow particular privilege attributes to be specified. The attributes supported by different systems may vary according to security policies supported. It is recommended that use of these interfaces be limited, so business application objects are not exposed to particular policy details (unless they need to be, as they are enforcing compatible security policies directly).

15.5.5 Operations on Object Reference

15.5.5.1 Description of Facilities

If the client application is unaware of security (for example, was written to use an ORB without security), the ORB services will enforce the relevant security policies transparently to applications. As described elsewhere, the security enforced is specified by:

- The security policy set at the client by administrative action.
- The credentials used by the client.
- The security policy for the target object. Relevant security information about this is made available to the client in the target's object reference.

These policies include association options, any controls on whether this client can perform this operation on this target, and the quality of protection of messages.

The only visibility of security to most applications is that some operations will now fail because they would breach security controls.

An application client unaware of security can communicate with a security-aware one and vice versa.

A client application aware of security can also specify what security policy options it wants to apply when communicating with this target object by performing operations on the target object's reference and the binding object associated with it. The following operations are available on the target object reference.

- **get_policy** is used to find the policy of the specified type (including those relevant to security) for this object.
- **get_domain_managers** is used to obtain a list of domain managers that the given object is associated with.
- **set_policy_override** is used to set overrides of default policies on individual object references.

Note – Although these operations are on the target object reference, the scope of the operation's effect is the use of that reference itself, and not the object that it represents. That is, the act of obtaining a copy of an object reference with a new set of override policies set on it in no way affects the target object that the object reference in question is associated with.

A target object can influence the security policy for incoming invocations by setting security policies using the administrative operations in Section 15.6, "Administrator's Interfaces," on page 15-130. This will affect the security information exported as part of its object reference.

The default policies that can be overridden using the **set_policy_overrides** operation are:

- **QOP** - the quality of protection that will be provided to any successful invocation using that object reference. The **QOPPolicy** object is the bearer of this policy.
- **Invocation Credentials** - the Credentials that will be used in invocations using that object reference. The **InvocationCredentialsPolicy** object is the bearer of this policy.
- **Security Mechanisms** - the mechanisms (one of) which must be used for successful invocation using the object reference. The **MechanismsPolicy** object is the bearer of this policy.

In order to set overrides, policy objects representing the new policies to be set must be created using the factory operations provided for doing so in the **SecurityLevel2::Current** interface. The set new policies to be used in **set_policy_overrides** must be put in a **PolicyList** to pass to the operation, which will return a new object reference with the new policy overrides set in it, if successful.

The policies currently associated with the object reference, including overridden ones can be accessed using the **get_policy** operation. This operation returns a Policy object of the appropriate type containing the current policy, which can be extracted from the readonly attribute in the Policy object interface. In addition to the standard policies there is an additional policy object (**SecurityFeaturesPolicy**) provided to enable access to the Security Features that are currently associated with the object reference. This object is used in conjunction with the **get_policy** operation, and is not a legal policy object to give as input to the **set_policy_overrides** operation.

Note – The application states its minimum security requirements. A higher level of security may still be enforced as this may be required by security policy. Thus, operationally, the default policies will actually be overridden only if the requested overrides are consistent with the overall security policy.

15.5.5.2 Client-Side Invocation Policy Objects

There are four Policy objects that are bearers of the client-side invocation-related policies. They are as follows:

QOP Policy

The QOP Policy object has the **QOPPolicy** interface shown below.

```
interface QOPPolicy : CORBA::Policy {          // Locality-Constrained
    readonly attribute Security::QOP                    qop;
};
```

This interface has a single readonly attribute **qop** which represents the policy in the form of an enum value of type **Security::QOP**.

This object can be passed to **set_policy_overrides** to specify that a particular quality of protection is required for messages sent using the object reference returned by the **set_policy_overrides** operation. When this object is returned by the **get_policy** operation it contains the quality of protection policy associated with this object reference.

Mechanism Policy

The Mechanism Policy object has the **MechanismPolicy** interface shown below:

```
interface MechanismPolicy : CORBA::Policy {// Locality-Constrained
    readonly attribute Security::MechanismTypeList  mechanisms;
};
```

This interface has a single readonly attribute **mechanisms** which represents the policy in the form of a **Security::MechanismTypeList**.

This object can be passed to **set_policy_overrides** to request the use of one of a specific set of mechanisms in invocation through the object reference returned by the **set_policy_overrides** operation. When this object is returned by **get_policy** it contains the security association mechanisms available through this object reference.

Invocation Credentials Policy

The Invocation Credentials Policy object has the **InvocationCredentialsPolicy** interface shown below:

```
interface InvocationCredentialsPolicy : CORBA::Policy {
                                                // Locality- Constrained
    readonly attribute CredentialsList                  creds;
};
```

This interface has a single readonly attribute **creds** which returns a list of **Credentials** objects which will be used as invocation credentials for invocations through this object reference.

This object can be passed to **set_policy_overrides** to specify one or more **Credentials** objects to be used when calling this target object using the object reference returned by **set_policy_overrides**. For example, the client may want to make different privileges available to different targets by choosing **Credentials** with the required privileges. When this object is returned by **get_policy** it contains the active credentials that will be used for invocations via this target object reference.

Security Features Policy

The Security Features Policy object has the **SecurityFeaturesPolicy** interface as shown below:

```
interface SecurityFeaturesPolicy : CORBA::Policy {// Locality-Constrained
    readonly attribute Security::SecurityFeatureValueList features;
};
```

This interface has a single readonly attribute **features** which returns a sequence of type **Security::SecurityFeatureValueList** containing the features currently associated with this object reference.

This object cannot be passed to **set_policy_overrides**. When this object is returned by the **get_policy** operation it contains the quality of protection and other security features which will apply to invocations via this object reference. It is a more efficient way of obtaining all the policy information related to security that is associated with this object reference.

15.5.5.3 Security-Relevant Operations in the CORBA::Object Interface

These operations are defined in detail in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*. A brief description is included here to help users of the Security Services.

get_policy

This gets the security policy object of the specified type, which applies to this object. This operation is also available on **Current** and is generally used there to get the policies for the current object.

The **get_policy** operation is used on object references during administration. For example, it may be used to get the policy for a domain.

```
CORBA::Policy get_policy(
    in      CORBA::PolicyType       policy_type
);
```

Parameters

policy_type	The type of policy to be obtained.

Return Value

policy	A policy object of the type specified by the policy_type parameter.

Exceptions

CORBA::BAD_PARAM	Raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

get_domain_managers

Allows administration services (and applications) to retrieve the domain managers, and hence the security and other policies applicable to individual objects that are members of the domain.

```
DomainManagersList get_domain_managers ();
```

Parameters

None.

Return Value

A list of immediately enclosing domain managers of this domain manager. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

set_policy_overrides

Makes it possible to override a subset of the policies that apply to a specific object reference. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **CORBA::SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**CORBA::ADD_OVERRIDE**) in the object reference, or they should be added to a clean override-free object reference (**CORBA::SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. The associations last until the object reference is destroyed or the process/capsule/ORB instance in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with current is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt will fail and return a CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

    Object set_policy_overrides(
        in      PolicyList              policies,
        in      SetOverrideType         set_add
);
```

Parameters

policies	A sequence of Policy objects that are to be associated with the new copy of the object reference returned by this operation.
set_add	Whether the association is in addition to (**ADD_OVERRIDE**) or as replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

15.5.5.4 Portability Implications

The security features that can be set are generally ones supported by a variety of security mechanisms. Applications using them will therefore be portable between any systems where the security mechanisms support these features. However, some security mechanisms will not support all features. For example, they may not provide replay protection, or may not support confidentiality of application data (owing to regulatory controls). Applications should check the response when attempting to set security features, and if a requested feature is not available, take suitable action.

15.5.6 Security Operations on Current

15.5.6.1 Description

The **Current** object represents service-specific state information associated with the current execution context (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*); both clients and targets have **Current** objects representing state associated with the thread of execution and the process/capsule in which the thread is executing (their execution contexts).

The operations of the **Current** object is intended to return information pertaining to the state associated with the current execution context. This includes information specific to both the thread of execution that is used to invoke the operation, as well as the process or capsule to which the thread belongs. State changes affecting state that is associated purely with the thread and not with any broader execution context like capsule (i.e., thread-specific) is lost, once the operation within the execution of which this was done completes its execution, thus returning the thread to the ORB. State changes to state associated with a broader execution context like a capsule (i.e., capsule-specific) on the other hand persists across multiple invocation of operations in the target object, until it is further modified through operations of the **Current** object or by other means.

The **SecurityLevel1::Current** and the **SecurityLevel2::Current** interfaces described in this section contain operations of both types. In this section, each operation is identified to be either thread-specific or process-specific to distinguish their behavior.

Note that a reference to the **Current** object representing the active execution context can be retrieved using the **ORB::resolve_initial_references("SecurityCurrent")** operation (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*). In a secure ORB, the **Current** object includes operations relevant to Security. The **CORBA::Current** object returned by the **resolve_initial_references** operation can be narrowed to **SecurityLevel1::Current** or **SecurityLevel2::Current** as desired.

These operations are described in this section and provide access to information about one or more of the following credentials:

- **invocation credentials**: These are the credentials at the client, used when this client invokes another object. There must always be credentials available for invocations, but setting these is generally done transparently to the business applications. When a user logs on, the user sponsor or other logon program normally sets this to the user's credentials. If this is done outside the object system, it is picked up at ORB initialization. At an intermediate object, its default value is either the received credentials or the object's own credentials, depending on the delegation policy that applies to that object.

- **own credentials**: the credentials associated with the active object. A particular object may have its own credentials or may share credentials with other objects. An object's own credentials are normally set up as the result of the object (or the environment domain to which it belongs) being initialized.
- **received credentials**: the credentials received from the client of the invocation as seen at the target object. Depending on delegation options, this may be a single **Credentials** object, or a list of credentials including those of both the initiator and other principals in the chain.
- **non-repudiation credentials**: when non-repudiation is supported, the credentials of the initiating principal in whose name evidence is being generated or verified. On logon, or ORB initialization, these are normally set to the user's credentials. At other objects, they are set by default to their own credentials.

The operations provided are the following:

- **get_attributes** (thread-specific) obtain privilege and other attributes associated with received credentials (which should be the user's privileges when at the workstation).
- **received_credentials** (thread-specific) attribute contains the credentials received from the client.
- **received_security_features** (thread-specific), an attribute at the target application that contains the security features of the message sent by the client.
- **set_credentials** (thread-specific) can specify the type of credentials. This changes the credentials to be used in the future for invocation, as its own credentials, or for non-repudiation. These **Credentials** apply only to those object references in which the invocations credentials have not been overridden.
- **get_credentials** (thread-specific) can obtain the credentials currently associated with the **Current** object for invocation, non-repudiation. These credentials apply to those object references in which the invocation credentials have not been overridden.

The application can also use the following:

- **get_policy** (capsule-specific) operation to find what security policies apply to it.
- **own_credentials** (capsule-specific) attribute containing the credentials owned by the application.
- **required_rights_object** (capsule-specific) attribute to discover which operations require which rights.
- **principal_authenticator** (capsule-specific) attribute to get a reference to the **PrincipalAuthenticator** object (which can be used to authenticate principals and thus obtain **Credentials** objects for them).
- **access_decision** (capsule-specific) attribute to get a reference to the **Access Decision** object.
- **audit_decision** (capsule-specific) attribute to get a reference to the **Audit Decision** object.
- **get_security_names** returns the security name(s) for the target.

It should be noted that when an application starts its execution and gains access to an ORB using the **ORB_init** operation, it immediately gets a set of default policies and credentials associated with it. Each thread executing in that capsule inherits these defaults and continues to be guided by them until any of the defaults are replaced by the use of a **set_*** operation in the thread. Subsequently the new credentials and features set using the **set_*** operation remain active until they are modified again by further use of **set_*** operations or the thread terminates or leaves the capsule. The corresponding **get_*** operations return the currently active credentials, policies, features, and attributes associated with the thread.

It should further be noted that if the policies associated with any individual object reference has been overridden using the **Object::set_policy_overrides** operation, then the overridden policies take precedence over the corresponding thread policies, when the said thread is used to carry out an object invocation using the said object reference.

In addition to the operations enumerated above factory operations for creating local invocation policy objects that are also provided in **Current**.

- **create_qop_policy**
- **create_mechanism_policy**
- **create_invoc_creds_policy**

15.5.6.2 The SecurityLevel1::Current Interface

The following operations are available in the **SecurityLevel1::Current** interface.

get_attributes

This is a thread-specific operation that is used to get privilege (and other) attributes from the client's credentials. It is available in the security functionality Level 1 to allow applications to enforce their own security policies without these applications having to perform operations on credentials.

This operation can be used to get:

- Privilege attributes for use in access control decisions. If the principal was not authenticated, only one privilege attribute is returned. This has type **Public** and no meaningful value.
- Other attributes, such as audit or charging identities, if available.

At the client, this generally gets the user's (or other principal's) privileges. At the target, it gets the received privileges.

```
AttributeList get_attributes(
    in      AttributeTypeList           attributes
);
```

Parameters

attributes	The set of security attributes (privilege attributes and identities) whose values are desired. if this list is empty, all attributes are returned.

Return Value

The set of attributes or identities reflecting the state of the **Credentials**.

15.5.6.3 The SecurityLevel2::Current Interface

The following operations are found in the **SecurityLevel2::Current** interface.

set_credentials

This operation pertains to the thread-specific state associated with the **Current** object. **Credentials** are associated with **Current** for different types of use. **Credentials** are automatically associated with **Current** by the object system at initialization, authentication, and object invocation. However, the application may want to specify particular credentials to use. The **set_credentials** operation sets the specified credentials as the default ones for the following:

- Subsequent invocations made by that client. (**SecInvocationCredentials**)
  This may be done to reduce the privileges available to that client by setting credentials having fewer privileges. Also, an intermediate object can explicitly ask for the received credentials to be delegated by using the **Current::received_credentials** as the specified credentials on **set_credentials**.
- Non-repudiation. (**SecNRCredentials**)
  As for the invocation credentials, non-repudiation credentials may be set transparently to the business application. The credentials used for non-repudiation may be the same as the credentials used for invocations. Note that in the previous sentence the word "credentials" is used in the English sense of the word and it does not refer to a **Credentials** object.

```
void set_credentials(
    in      CredentialType        cred_type,
    in      CredentialsList       creds,
    in      DelegationMode        del
);
```

Parameter

cred_type	The type of credential to be set (i.e., **SecInvocationCredentials** or **SecNRCredentials**).
creds	The object reference of the **CredentialsList**, which is to become the default
del	The delegation mode for the credentials being set takes values of **Delegate** or **NoDelegate**.

Return Value

None

get_credentials

This thread-specific operation allows an application access to the credentials associated with its execution environment. As for **set_credentials**, the application can ask for the default credentials for future invocations or the ones used for non-repudiation.

An application will normally get invocation or other credentials when it wants to modify them (for example, reduce the privileges available).

```
CredentialsList get_credentials(
    in      CredentialType        cred_type
);
```

Parameters

cred_type	The type of credential to be obtained.

Return Value

A **CredentialsList**.

received_credentials

At a target object, this thread-specific operation gets the credentials received from the client. If credentials representing more than one principal are received, the contents of these credentials depend on the delegation model in use. They may be:

- The credentials of the only principal identified, if simple delegation is used (or if the security technology used has merged the credentials of all the callers in the chain).
- A list of credentials, if the credentials for different principals in a chain of calls can be distinguished. Note that the number of credentials in this list depends on the delegation option in use. There may be credentials for the initiator of the chain and the immediate invoker only, or credentials providing a trace of all principals in the

chain. The first entry in the chain is the "primary" principal's credentials, normally the credentials of the initiator of the chain. A **Current::get_attributes** returns the privileges from these credentials.

At the user agent client, the **received_credentials** attribute is the user's credentials.

readonly attribute CredentialsList received_credentials;

Return Value

A sequence of **Credential** object references received from the requester.

own_credentials

Any application owns a set of credentials which it obtains through the process of authentication of the principal that initiates the execution of the program, and further from other credentials that such a principal might bestow upon the application. This attribute returns this set of credentials.

readonly attribute CredentialsList own_credentials;

Return Value

A sequence of **Credentials** object references owned by the application.

received_security_features

This thread-specific attribute at the target application provides the security features of the message sent by the client.

readonly attribute SecurityFeatureValueList received_security_features;

Return Value

A sequence of feature-value pairs. A boolean value of TRUE indicates that the feature is on; otherwise FALSE.

get_policy

This capsule-specific operation returns the policy object of the specified policy_type for the non CORBA object client from which it is invoked, or for the CORBA object from which it is invoked.

```
Policy get_policy(
    in CORBA::PolicyType      policy_type
);
```

Parameters

policy_type	The type of policy to be obtained.

Return Value

A policy object which can be used to interrogate the policy in force as defined in Section 15.6, "Administrator's Interfaces," on page 15-130. For example, the secure invocation policy would give the secure associations defaults for this object, and the delegation policy would say which credentials were delegated on invocations by this object.

required_rights_object

This capsule-specific read-only attribute is the **RequiredRights** object available in the environment. This object is rarely used directly by applications. It is generally used by **Access Decision** objects to find the rights required to use a particular interface; however, it could be used directly by the application if it wishes to do all its own access control, and base this on **Rights**.

readonly attribute RequiredRights required_rights_object;

Return Value

An object references to a **RequiredRights** object. The operations in the interface of this object are defined in Section 15.6.4, "Access Policies," on page 15-133.

principal_authenticator

This capsule-specific read-only attribute is the **PrincipalAuthenticator** object available in the environment. It can be used by the application to authenticate principals and obtain **Credentials** containing their privilege attributes.

readonly attribute PrincipalAuthenticator principal_authenticator;

Return Value

An object references to a **PrincipalAuthenticator** object. The operations in the interface of this object are defined in Section 15.3.2, "Principals and Their Security Attributes," on page 15-20.

access_decision

This capsule-specific read-only attribute is the **AccessDecision** object available in the environment. It can be used by the application to obtain decisions regarding accessibility of specific objects from this environment.

readonly attribute AccessDecision access_decision;

Return Value

An object references to an **AccessDecision** object. The operations in the interface of this object are defined in Section 15.5.9, "Access Control," on page 15-117.

audit_decision

This capsule-specific read-only attribute is the **AuditDecision** object available in the environment. It can be used by the application to obtain information about what needs to be audited for the specified object/interface in this environment.

```
readonly attribute AuditDecision audit_decision;
```

Return Value

An object references to an **AuditDecision** object. The operations in the interface of this object are defined in Section 15.5.7, "Security Audit," on page 15-113.

get_security_names

This operation is for use by security sophisticated applications. It is used by clients who wish to determine which security names are associated with the target. It is possible for different security names to be used for the target, depending on the mechanism used for the target. The name may be shared by several objects.

```
SecurityMechandNameList get_security_names(
    in        Object                        )obj_ref
),
```

Parameters

obj_ref	The Object reference of the target object of which the security names are being sought.

Return Value

A list of pairs of values, each containing a security mechanism and associated security name.

create_qop_policy

This operation is used to create a local **QOP Policy** object.

```
QOPPolicy create_qop_policy(
    in        Security::QOP                 qop
);
```

Parameters

qop	A quality of protection specification.

Return Value

An object reference to a new **QOP Policy** object containing the protection value specified in **qop**.

create_mechanism_policy

This operation is used to create a local **Mechanism Policy** object

```
MechanismsPolicy create_mechanism_policy(
    in      Security::MechanismTypeList     mechanisms
);
```

Parameters

mechanisms	A list of mechanisms that are acceptable.

Return Value

An object reference to a new **Mechanism Policy** object containing the mechanisms list given in the **mechanism** parameter.

create_invoc_creds_policy

This operation is used to create a local **Invocation Credentials Policy** object.

```
InvocationCredentialsPolicy create_invoc_creds_policy(
    in      CredentialsList                 creds
);
```

Parameters

creds	A list of **Credentials**.

Return Value

An object reference to a new **Invocation Credentials Policy** object containing the **Credentials** passed through the **creds** parameter.

15.5.7 Security Audit

15.5.7.1 Description of Facilities

Auditing of object invocations is done automatically by the ORB according to the audit invocation policies (**Security::SecClientInvocationAudit** and **Security::SecTargetInvocationAudit**) for this application.

Applications can also audit their own security-relevant activities, where the auditing performed by the ORB does not audit the required activities and/or data.

In this case, the application is responsible for enforcing the application audit policy. It uses an **audit_needed** operation on the **Audit Decision** object for the policy to decide which activities to audit.

Audit information is passed to an **Audit Channel** object in the form of an audit record. The audit record must contain, or be sufficient to identify:

- The type of event.
- The principal responsible for the action, identified by its credentials.
- Event-specific data associated with the event type. This will vary, depending on the event type.
- The time. This may or may not be secure.

It may also want to record some of the values used for selecting whether to audit the event, for example, its success or failure.

An application audit policy will specify the event families and event types as defined in Section 15.6.5, "Audit Policies," on page 15-144.

15.5.7.2 The SecurityLevel2::AuditDecision Interface

The **Audit Decision** object has the **SecurityLevel2::AuditDecision** interface. Its operations described below, help specify what to audit. It is a **locality-constrained** object.

The **Audit Decision** object is a **locality-constrained** object.

audit_needed

This operation on the **Audit Decision** object is used to decide whether an audit record should be written to the audit channel. The application specifies the event type to be checked and the values for the selectors, which the audit policy requires to make the decision.

```
boolean audit_needed(
    in AuditEventType           event_type,
    in SelectorValueList        value_list
);
```

Parameters

event_type	Event type associated with the operation.
value_list	List of zero or more selector id value pairs.

Return Value

TRUE	If an audit record should be created and sent to the audit channel.
FALSE	If an audit record is not needed.

audit_channel

This attribute of the **Audit Decision** object provides the audit channel associated with this audit decision object.

```
readonly attribute AuditChannel audit_channel;
```

Return Value

The **Audit Channel** object associated with the **Audit Decision** object.

A standard audit policy is specified in Section 15.6, "Administrator's Interfaces," on page 15-130, but if this is to be replaceable without ORB/interceptor changes, a standard interface needs to be available for the ORB or interceptor to call. Therefore, for replaceability, the selectors used on audit needed during invocation must always be the same (see **value_list** above), though not all of these need to be used in taking the decision to audit, depending on policy. Note that the time is not passed over this interface. If the selectors specified in the audit policy use time to decide on whether to audit the event, the **Audit Decision** object should obtain the current time itself.

15.5.7.3 The SecurityLevel2::AuditChannel Interface

The single operation in the **SecurityLevel2::AuditChannel** interface is used to write the audit records. The **Audit Channel** object is a **locality-constrained** object.

audit_write

This operation writes an audit record to the **Audit Channel** object, and hence the audit trail. The audit trail is implementation-specific and outside the scope of this chapter. It is expected to be an event service of some sort, such as a CORBA Event Service.

```
void audit_write(
    in      AuditEventType          event_type,
    in      CredentialsList         creds,
    in      UtcT                    time,
    in      SelectorValueList       descriptors,
    in      Opaque                  event_specific_data
);
```

Parameters

event_type	The type of event being audited.
creds	The credentials of the principal responsible for the event. If no credentials are specified, the **own_credentials** attribute associated with **Current** are used.

time	The time the event occurred.
descriptors	A set of values to be recorded associated with the event in the audit trail. These are often the same values as those used to select whether to audit the event.
event_specific_data	Data specific to a particular type of event, to be recorded in the audit trail.

Return Value

None.

audit_channel_id

A readonly attribute that contains the id of this audit channel, which is used to identify it in the corresponding audit policy object. This is necessary because the audit channel object itself has to be a **locality-constrained** object by virtue of the fact that the **audit_write** operation passes a list of **Credentials**, a **locality-constrained** object, as a parameter, while the audit policy object needs to be not thus constrained.

The audit channel identified by the **audit_channel_id** in the **Audit Policy** object is actually associated with the **Audit Channel** interface by the **Audit Decision** object when its **audit_channel** attribute is accessed.

```
readonly attribute AuditChannelId audit_channel_id;
```

Return Value

audit_channel_id	The channel id of the audit channel.

15.5.7.4 Portability Implications

An application relying on the system audit policies enforced at invocation time is portable to different environments, although the audit policies themselves may need changing.

Applications with their own application audit policies are portable, providing the audit policy itself is portable and the selectors used are available in these environments. For example, if selectors use privileges, the same ones must be available.

15.5.8 Administering Security Policy

When an object is created, it automatically becomes a member of one or more domains, and therefore is subject to the security policies of those domains.

Security-aware applications can administer security policies (providing they are authorized to do so) using the interfaces described in Section 15.6, “Administrator’s Interfaces,” on page 15-130.

15.5.9 Access Control

15.5.9.1 Description of Facilities

Access policies for applications may be enforced in the following ways:

- Automatically by the ORB services on object invocation, to determine whether the caller has the right to invoke an operation on an object.
- By the application itself, to enforce further controls on who can invoke it to do what.
- By the application to control access to its own internal functions and state.

This section is concerned with applications that wish to enforce their own access controls, either supplementing the automatic controls on invocation or controlling internal functions.

As explained in Access Policies under Section 15.3.4, "Access Control Model," on page 15-24, the decision on whether to allow such access may use the following:

- The principal's credentials (which either contain its privilege attributes, or identify the principal so these can be obtained). Using only the principal's identity generally requires that identity to be known at all targets, and leads to scalability problems, so its use is depreciated. Use of the principal's role or group(s) are more likely to give easier administration in large systems, as would security clearance. Enterprise-defined attributes can also be used when supported.
- The target's control attributes such as an ACL or security classification.
- Other relevant information about the action such as the operation (on object invocation) and parameters, and also context information such as time.
  The application can use rights associated with an interface (as described in Section 15.6.3, "Security Policies Introduction," on page 15-132) rather than specify controls for individual operations.
- The security policy rules using this information as enforced by the access decision function.

The access policies enforced automatically by the ORB during object invocation can take into account the principal's credentials, the target's control attributes, the operation and the time (though the time is not used in the standard access policy defined in Section 15.6, "Administrator's Interfaces," on page 15-130). However, the ORB does not use the parameters to the operation for controlling access. So, for example, if there is a rule that only senior managers can authorize expenditure over $5000, the application is likely to need its own function to perform the required check.

Where an application enforces its own access decisions, it will be responsible for maintaining its own control information about operations, functions, and data it wishes to protect. It can do this in a way specific to its own particular functions or data, but in some cases, it is possible to have a more generic way of handling access decisions, and in these cases, it may be possible to use a common access decision object with common administration of the ACLs or other control attributes.

15.5.9.2 The Access Decision Object

The access decision functionality is encapsulated in **Access Decision** objects. These may require different information depending on, for example, the action or data to be controlled and the security policy rules as previously described. The **Access Decision** object is a **locality-constrained** object.

The **Access Decision** object has the **access_allowed** operation as is used for enforcing access policies in the ORB (see below). The input parameters to this should normally specify:

- The privileges of the initiator of the action. The form of these depends on the specific policy. Some options are:
  - The privileges of the initiator as supplied by a **get_attributes** operation on **Current** (see "The SecurityLevel1::Current Interface" on page 15-107).
  - A credentials object, which represents principal.
  - A credentials list (the **received_credentials**), where access controls distinguish initiator and delegate principals.
- Other information required by the access decision function, including:
  - Application-level decisions on whether an invocation is permitted, the operation and parameters passed in the request, and the object reference.
  - Control of access to internal functions and data, the action, and relevant parameters.

The return value from the **access_allowed** operation is either TRUE signifying access is permitted, or FALSE signifying that it is not

It is recommended that where possible, access decisions are made by such **Access Decision** objects (or at least separate internal functions) that hide details of the actual security policy used, so the application does not need to know, for example, whether an ACL or label-based policy is used.

15.5.9.3 The SecurityLevel2::AccessDecision Interface

The **Access Decision** object is a **locality-constrained** object. The **AccessDecision** interfaces have the following single operation:

access_allowed

```
boolean access_allowed(
    in      SecurityLevel2::CredentialsList     cred_list,
    in      Object                              target,
    in      CORBA::Identifier                   operation_name,
    in      CORBA::Identifier                   target_interface_name
);
```

Parameters

cred_list	The list of **Credentials** associated with the request. The list may be empty (in the case of unauthenticated requests), it may contain only a single credential, or it may contain several credentials (in the case of delegated or otherwise cascaded requests). The **Access Decision** object is presumed to have rules for dealing with all these cases.
target	The reference used to invoke the target object. The method invoked.
operation_name	The name of the operation being invoked on the target.
target_interface_ name	The name of the interface to which the operation being invoked belongs. This may not be required in some implementations and will only be required in cases in which the operation being invoked does not belong to the interface of which the target object is a direct instance.

Return Value

boolean	A return value of TRUE indicates that the request should be allowed, otherwise FALSE**.**

15.5.9.4 Portability Implications

Portability of applications enforcing their own access controls is improved by use of **Access Decision** objects as previously described. The application then does not need to know the particular rules used, and even which principal and object attribute types are used to decide whether access should be permitted. It can also hide whether the principal's credentials include all privilege attributes needed, or whether these are obtained dynamically when needed.

Different systems may need to support different access control policies. By hiding details of the access control rules used to enforce the policy behind a standard interface, the application will generally be portable to environments with different policies.

Applications that use their own specific code to make access decisions will only be portable to systems that support the identity and privilege attribute types used in those decisions with the same syntax.

15.5.10 Delegation Facilities

15.5.10.1 Description of Facilities

An operation on a target object may result in calls on many other objects as described in Section 15.3.6, "Delegation," on page 15-30. An intermediate object in this chain of objects may:

- Delegate the credentials received (often containing the initiating principal's privileges) to the next object in the chain, so access decisions at the target may be based on that principal's privileges.
- Act on its own behalf, so use its own credentials when invoking another object in the chain.
- Supply privileges from both, so access decisions at the target object can take into account both the initiating principal's privileges and where these came from.

Which of these delegation modes should be used depends on the application. For example, a user might call a database object asking for some data, and this may obtain the data from a file that also contains data belonging to other users. In this example, the database object would control access to the data using the user's privileges, whereas the filestore object would use the database's privileges.

In general, the delegation mode used is specified by the administrator in the delegation policy for objects of this type in this domain. However, a security-aware application can also specify the delegation mode it wants to use, as it may want different modes when invoking different objects.

15.5.10.2 Operations

All the operations used for delegation are specified elsewhere. This section describes how they are used during delegation.

An intermediate object can set the delegation mode for an invocation by performing a **Credentials::set_security_features** operation of the credentials object to be used for the invocation (see Section 15.5.4, "The Credentials Object," on page 15-94). This can be used to set the delegation mode to:

- **SecNoDelegation**, meaning use the intermediate's object's own credentials.
- **SecSimpleDelegation**, meaning use the credentials received from the client.
- **SecCompositeDelegation**, meaning use both.

The way the received and intermediate's own credentials are combined in **SecCompositeDelegation** is not defined. Depending on the implementation:

- The initiating principal's and the intermediate's own credentials are passed, and are available separately at the target.
- The received credentials and intermediate's own credentials are combined, so the target sees only a single credentials object with privileges from each of these.

- **Credentials** from all objects in the delegation chain are passed and are available separately to the target.

None of these particular composite delegation modes are part of the Security Functionality Level 2. They are described here because of the effect on the **Current::received_credentials** (see “The SecurityLevel2::Current Interface” on page 15-108), which a target object uses to find out who called it. The target normally uses this to get privileges for use in access control decisions.

The **Current::received_credentials** attribute provides a **CredentialsList**, not just a single **Credentials** object. This list will only have more than one entry after composite delegation as defined above. If there is more than one entry in the list, the first entry is that of the initiator in the chain, normally the main one used for access controls. This is also the one whose privileges are obtained via **Credentials::get_attributes**.

15.5.10.3 Portability Implications

Where possible, the delegation mode should be set using the administrative interfaces to the delegation policy, so applications may delegate privileges (or not) without any application-level code, and so be portable.

If an application sets its own delegation mode, it should be able to handle a CORBA::NO_IMPLEMENT exception if **SecCompositeDelegation** is specified, as this may not be supported.

If the application wants to enforce its own access policy, it should use an **Access Decision** object (as described in Section 15.5.9, “Access Control,” on page 15-117), which hides whether access decisions utilize the initiator’s privileges separately from the delegate’s privileges.

However, where an application wants to provide specific checks which intermediates have been involved in performing the original user’s operation, such checks are likely to depend on the delegation scheme and its implementation, and so not be portable.

15.5.11 Non-repudiation

Non-repudiation is an optional facility.

15.5.11.1 Description of Facilities

The Non-repudiation Service provides evidence of application actions in a form that cannot be repudiated later. This evidence is associated with some data (for example, the amount field of a funds transfer document).

Non-repudiation evidence is provided in the form of a token. Two token types are supported:

- Token including the associated data
- Token without included data (but with a unique reference to the associated data)

Non-repudiation tokens may be freely distributed. Any possessor of a non-repudiation token (and the associated data, if not included in the token) can use the non-repudiation Service to verify the evidence. Any holder of a non-repudiation token may store it (along with the associated data, if not included in the token) for later adjudication.

The non-repudiation interfaces support generation and verification of tokens embodying several different types of evidence. It is anticipated that the following will be the most commonly used non-repudiation evidence token types:

- Non-repudiation of Creation prevents a message creator's false denial of creating a message.
- Non-repudiation of Receipt prevents a message recipient's false denial of having received a message.

Generation and verification of non-repudiation tokens require as context a non-repudiation credential, which encapsulates a principal's security information (particularly keys) needed to generate and/or verify the evidence. Most operations provided by the Non-repudiation Service are performed on **NRCredentials** objects.

Non-repudiation Service operations supported by the **NRCredentials** interface are as follows.

- **set_NR_features** specifies the features to apply to future evidence generation and verification operations.
- **get_NR_features** returns the features which will be applied to future evidence generation and verification operations.
- **generate_token** generates a non-repudiation token using the current non-repudiation features. The generated token may contain:
  - Non-repudiation evidence.
  - A request, containing information describing how a partner should use the Non-repudiation Service to generate an evidence token.
  - Both evidence and a request.
- **verify_evidence** verifies the evidence token using the current non-repudiation features.
- **get_token_details** returns information about an input non-repudiation token. The information returned depends upon the type of the token (evidence or request).
- **form_complete_evidence** is used when the evidence token itself does not contain all the data required for its verification, and it is anticipated that some of the data not stored in the token may become unavailable during the interval between generation of the evidence token and verification unless it is stored in the token. The **form_complete_evidence** operation gathers the “missing” information and includes it in the token so that verification can be guaranteed to be possible at any future time.

  The **verify_evidence** operation returns an indicator (**evid_complete**), which can be used to determine whether the evidence contained in a token is complete. If a token’s evidence is not complete, the token can be passed to **form_complete_evidence** to complete it.

If complete evidence is always required, the call to **form_complete_evidence** can, in some cases, be avoided by setting the **form_complete** request flag on the call to **verify_evidence**; this will result in a complete token being returned via the **evid_out** parameter.

15.5.11.2 Non-repudiation Service Data Types

The following data types are used in the Non-repudiation Service interfaces:

```
module NRservice {
    typedef MechanismType        NRMech;
    typedef ExtensibleFamily     NRPolicyId;

    enum EvidenceType {
        SecProofofCreation,
        SecProofofReceipt,
        SecProofofApproval,
        SecProofofRetrieval,
        SecProofofOrigin,
        SecProofofDelivery,
        SecNoEvidence    // used when request-only token desired
    };

    enum NRVerificationResult {
        SecNRInvalid,
        SecNRValid,
        SecNRConditionallyValid
    };

    typedef unsigned long DurationInMinutes;

    const  DurationInMinutes     DURATION_HOUR = 60;
    const  DurationInMinutes     DURATION_DAY   = 1440;
    const  DurationInMinutes     DURATION_WEEK = 10080;
    const  DurationInMinutes     DURATION_MONTH = 43200;// 30 days
    const  DurationInMinutes     DURATION_YEAR = 525600;//365 days

    typedef long TimeOffsetInMinutes;

    struct NRPolicyFeatures {
        NRPolicyId              policy_id;
        unsigned long           policy_version;
        NRMech                  mechanism;
    };

    typedef sequence <NRPolicyFeatures> NRPolicyFeaturesList;

    // features used when generating requests
        struct RequestFeatures {
        NRPolicyFeatures        requested_policy;
```

```
        EvidenceType        requested_evidence;
        string              requested_evidence_generators;
        string              requested_evidence_recipients;
        boolean             include_this_token_in_evidence;
    };
};
```

15.5.11.3 The NRservice::NRCredentials Interface

This section describes the Non-repudiation Service operations that are provided by the **NRCredentials** interface.

set_NR_features

When an **NRCredentials** object is created, it is given a default set of NR features, which determine what NR policy will be applied to evidence generation and verification requests.

Security-aware applications may set NR features to specify policy affecting evidence generation and verification. The interface for setting NR features is:

```
boolean set_NR_features(
    in      NRPolicyFeaturesList        requested_features,
    out     NRPolicyFeaturesList        actual_features
);
```

Parameters

requested_features	The non-repudiation features required.
actual_features	The NR features that were set (may differ from those requested depending on implementation).

Return Value

TRUE	The requested features were equivalent.
FALSE	The actual features differ from the requested features.

get_NR_features

This operation is provided to allow security-aware applications to determine what NR policy is currently in effect:

```
NRPolicyFeaturesList get_NR_features ();
```

Parameters

None

Return Value

The current set of **NR features** in use in this **NRCredentials** object.

generate_token

This operation generates a non-repudiation token associated with the data passed in an input buffer. Environmental information (for example, the calling principal's name) is drawn from the **NRCredentials** object.

If the data for which non-repudiation evidence is required is larger than can conveniently fit into a single buffer, it is possible to issue multiple calls, passing a portion of the data on each call. Only the last call (i.e., the one on which **input_buffer_complete = true**) will return an output token and (optionally) an evidence check.

```
void generate_token(
    in      Opaque             input_buffer,
    in      EvidenceType       generate_evidence_type,
    in      boolean            include_data_in_token,
    in      boolean            generate_request,
    in      RequestFeatures    request_features,
    in      boolean            input_buffer_complete,
    out     Opaque             nr_token,
    out     Opaque             evidence_check
);
```

Parameters

input_buffer	Data for which evidence should be generated.
generate_evidence_type	Type of evidence token to generate (may be **SecNoEvidence**).
include_data_in_token	If set TRUE, data provided in **input_buffer** will be included in generated token; otherwise FALSE.
generate_request	The output token should include a request, as described in the **request_features** parameter.
request_features	A structure describing the request. Its fields are listed below:

requested_policy	Non-repudiation policy to use when generating evidence tokens in response to this request
requested_evidence	Type of evidence to be generated in response to this request.
requested_evidence_ generators	Names of partners who should generate evidence in response to this request.
requested_evidence_ recipients	Names of partners to whom evidence generated in response to this request should be sent.
include_this_token_ in_evidence	If set true, the evidence token incorporating the request will be included in the data for which partners will generate evidence. If set false, evidence will be generated using only the associated data (and not the token incorporating the request).
input_buffer_complete	True if the contents of the input buffer complete the data for which evidence is to be generated; false if more data will be passed on a subsequent call.
nr_token	The returned NR token.
evidence_check	Data to be used to verify the requested token(s) (if any) when they are received.

Return Value

None.

verify_evidence

Verifies the validity of evidence contained in an input NR token.

If the token containing the evidence to be verified was provided to the calling application by a partner responding to the calling application's request, then the calling application should pass the evidence check it received when it generated the request as a parameter to **verify_evidence** along with the token it received from the partner.

It is possible to request the generation of complete evidence. This may succeed or fail; if it fails, a subsequent call to **form_complete_evidence** can be made. Output indicators are provided, which give guidance about the time or times at which **form_complete_evidence** should be called; see the parameter descriptions for explanations of these indicators and their use. Note that the time specified by

complete_evidence_before may be earlier than that specified by **complete_evidence_after**; in this case it will be necessary to call **form_complete_evidence** twice.

Because keys can be revoked or declared compromised, the return from **verify_evidence** cannot in all cases be a definitive "**SecNRValid**" or "**SecNRInvalid**"; sometimes "**SecNRConditionallyValid**" may be returned, depending upon the policy in use. "**SecNRConditionallyValid**" will be returned if:

- The interval during which the generator of the evidence may permissibly declare his key invalid has not yet expired (and therefore it is possible that the evidence may be declared invalid in the future), or
- Trusted time is required for verification, and the time obtained from the token is not trusted.

```
NRVerificationResult verify_evidence(
    in      Opaque          input_token_buffer,
    in      Opaque          evidence_check,
    in      boolean         form_complete_evidence,
    in      boolean         token_buffer_complete,
    out     Opaque          output_token,
    out     Opaque          data_included_in_token,
    out     boolean         evidence_is_complete,
    out     boolean         trusted_time_used,
    out     TimeT           complete_evidence_before,
    out     TimeT           complete_evidence_after
);
```

Parameters

input_token_buffer	Buffer containing (possibly a portion, possibly all of) evidence token to be verified; buffer may also contain data associated with evidence token (parsing of buffer in this case is understood only by NR mechanism, see **get_token_details**).
evidence_check	The evidence check.
form_complete_ evidence	Set TRUE if complete evidence is required; otherwise FALSE.
token_buffer_ complete	Set TRUE if the **input_token_buffer** completes the input token; FALSE if more input token data remains to be passed on a subsequent call.
output_token	If **form_complete_evidence** was set to TRUE, this parameter will contain complete evidence (and the Return Value will be **SecNRValid**) or an "augmented" but still incomplete evidence token, in which case **SecNRConditionallyValid** is returned

data_included_in_ token	Data associated with the evidence, extracted from input token (may be null).
evidence_is_ complete	TRUE if evidence in input token is complete, otherwise FALSE.
trusted_time_used	TRUE if the evidence token contains a time considered to be trusted according to the rules of the non-repudiation policy. FALSE indicates that the security policy mandates trusted time and that the time in the token is not considered to be trusted.
complete_evidence_ before	If **evidence_is_complete** is FALSE and the return value from **verify_evidence** is SecNRConditionallyValid, the caller should call **form_complete_evidence** with the returned output token before this time. This may be required, for example, in order to ensure that the time skew between the evidence generation time and the trusted time service's countersignature on the evidence falls within the interval allowed by the current NR policy.
complete_evidence_ after	If **evidence_is_complete** is FALSE and the return value from **verify_evidence** is **SecNRConditionallyValid**, the caller should call **form_complete_evidence** with the returned output token after this time. This may be required, for example, to ensure that all authorities involved in generating the evidence have passed the last time at which the current NR policy allows them to repudiate their keys.

Return Value

SecNRInvalid	Evidence is invalid.
SecNRValid	Evidence is valid.
SecNRConditionallyValid	Evidence cannot yet be determined to be invalid

get_token_details

The information returned depends upon the type of the token (evidence or request). The mechanism that created the token is always returned.

- If the input token contains evidence, the following is returned: the non-repudiation policy under which the evidence has been generated, the evidence type, the date and time when the evidence was generated, the name of the generator of the evidence, the size of the associated data, and an indicator specifying whether the associated data is included in the token.

- If the input token contains a request, the following is returned: the name of the requester of the evidence, the non-repudiation policy under which the evidence to send back should be generated, the evidence type to send back, the names of the recipients who should generate and distribute the requested evidence, and the names of the recipients to whom the requested evidence should be sent after it has been generated.
- If the input token contains both evidence and a request, an indicator describing whether the partner's evidence should be generated using only the data in the input token, or using both the data and the evidence in the input token.

```
void get_token_details(
    in      Opaque              token_buffer,
    in      boolean             token_buffer_complete,
    out     string              token_generator_name,
    out     NRPolicyFeatures    policy_features,
    out     EvidenceType        evidence_type,
    out     UtcT                evidence_generation_time,
    out     UtcT                evidence_valid_start_time,
    out     DurationInMinutes   evidence_validity_duration,
    out     boolean             data_included_in_token,
    out     boolean             request_included_in_token,
    out     RequestFeatures     request_features
);
```

Parameters

token_buffer	Evidence token to parse.
token_buffer_complete	Indicator when the token has been fully provided.
token_generator_name	Principal name of token generator.
policy_features	Describes the policy used to generate the token.
evidence_type	Type of evidence contained in the token (may be **SecNoEvidence**).
evidence_generation_time	Time when evidence was generated.
evid_validity_start_time	Beginning of evidence validity interval.
evidence_validity_duration	Length of evidence validity interval.
data_included_in_token	TRUE if the token includes the data for which it contains evidence, otherwise FALSE.
request_included_in_token	TRUE if the token includes a request, otherwise FALSE.
evidence_generation_time	Time when evidence was generated.

Return Value

None.

form_complete_evidence

form_complete_evidence is used to generate an evidence token that can be verified successfully with no additional data at any time during its validity period.

```
boolean form_complete_evidence(
    in      Opaque      input_token,
    out     Opaque      output_token,
    out     boolean     trusted_time_used,
    out     TimeT       complete_evidence_before,
    out     TimeT       complete_evidence_after
);
```

Parameters

token_buffer	Evidence token to be completed..
output_token	The “augmented” evidence token may be complete.
trusted_time_used	TRUE if the token’s generation time can be trusted, otherwise FALSE. If trusted time is required by the policy under which the evidence will be verified, and if this indicator is not set, the evidence will not be considered complete.
complete_evidence_before	If the return value is FALSE, **form_complete_evidence** should be called before this time.
complete_evidence_after	If the return value is FALSE, **form_complete_evidence** should be called after this time.

Return Value

TRUE	Evidence is now complete.
FALSE	Evidence is not yet complete.

15.6 Administrator’s Interfaces

This section describes the administrative features of the specification. Administration specifies the policies that control the security-related behavior of the system. These features form an ‘Administrator’s View,’ encompassing the interfaces that a human administrator would need to use, but the facilities may also be used by conventional applications that wish to be involved in administrative actions. ‘Administrator’ may therefore refer to a human or system agent.

Most interfaces defined here are in Security Functionality Level 2, as Level 1 security does not include administration interfaces.

15.6.1 Concepts

15.6.1.1 Administrators

This specification imposes no constraints on how responsibilities are divided among security administrators, but in many cases an enterprise will have a security policy that restricts the responsibilities of any one individual. Also, legal requirements may dictate a separation of roles so that, for example, there are different administrators for access control and auditing functions.

Administrators are subject to the same security controls as other users of the system. It is expected that an enterprise will define roles (or other privileges) that certain administrators will adopt. Administrative operations are subject to access controls and auditing in the same way as other object invocations, so only administrators with the required administrative privileges will be able to invoke administrative operations.

This specification does not define administrative functions concerning the management of underlying mechanisms supporting the security services, such as an Authentication Service, Key Distribution Service, or Certification Authority.

15.6.1.2 Policy Domains

Security administrators specify security policies for particular security policy domains (for brevity, only the words in bold are used for the remainder of this section).

A domain includes an object, called the domain manager, which has associated with it the policy objects for this domain, and notionally contains zero or more other objects, which are domain members and subject to the policies specified by the policy objects associated with the domain manager.

The domain manager records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

There are different types of policy objects for administering different types of policy. As described in "Security Policy Domains" on page 15-38, domains may be members of other domains, forming containment hierarchies. Because different kinds of policy affect different groups of objects, objects (and domains) may be members of multiple domains.

The policies that apply to an object are those of all its enclosing domains.

15.6.1.3 Security Policies

This specification covers administration of security policies, which are enforced by a secure object system in either of the following ways:

- Automatically on object invocation. This covers system policies for security communications between objects, control of whether this client can use this operation on this target object, whether the invocation should be audited, and whether an original principal's credentials can be delegated.
- By the application. This covers security policies enforced by applications. Applications may enforce access, audit, and non-repudiation policies. The application policies may be managed using domains as for other security policies, or the application can choose to manage its own policies in its own way.

Invocation time policies for an object can be applicable only when it is a target object, or whenever it is acting as either a target object or a client.

Security policies may be administered by any application with the right to use the security administrative interfaces. This is subject to the invocation access control policy for the administrative interface.

15.6.2 Domain Management

The Domain Management facilities (defined in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*) are used by the Security Service as described in the following sections.

15.6.3 Security Policies Introduction

Invocation security policies are enforced automatically by ORB services during object invocation. These are:

- Invocation access policies (**Security::SecClientInvocationAccess** and **Security::SecTargetInvocationAccess**, interface **SecurityAdmin::AccessPolicy**) for controlling access to objects.
- Invocation audit policies (**Security::SecClientInvocationAudit** and **Security::SecTargetInvocationAudit**, interface **SecurityAdmin::AuditPolicy**) control which operations on which objects are to be audited.
- Invocation delegation policies (**Security::SecDelegation**, interface **SecurityAdmin::DelegationPolicy**) for controlling the delegation of privileges.
- Secure invocation policies (**Security::SecClientSecureInvocation** and **Security::SecTargetSecureInvocation**, interface **SecurityAdmin::SecureInvocationPolicy**) for security associations, including controlling the delegation of client's credentials, and message protection.

Different policies generally apply when an object acts as a client from when it is the target of an invocation.

In addition to these invocation policies, there are a number of policy types, which apply independently of object invocation. These are:

- Application access policy (**Security::SecApplicationAccess**, interface **SecurityAdmin::AccessPolicy**), which applications may use to manage and enforce their access policies.

- Application audit policy (**Security::SecApplicationAudit**, interface **SecurityAdmin::AuditPolicy**), which applications can use to manage and enforce their audit policies.
- Non-repudiation policies (**Security::SecNonRepudiation**, interface **SecurityAdmin::NRPolicy**) determine the rules for the generation and use of evidence.

There is also a policy concerned with the creation of object references, which is enforced by **POA::create_reference** and variants thereof or equivalent operations. This is the construction policy (**CORBA::SecConstruction**) which controls whether a new domain is created when an object of a specified type is created. (See the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*.)

Note – Policies associated with underlying security technology are not included. For example, there are no policies for principal authentication as this is often done by specific security services.

Operations are provided for setting all the types of security policies previously listed. In each case, these management operations permit administration of standard policy semantics supported by the interfaces defined in this specification. It is also possible for implementors to replace the policy objects, the operations of which are defined in this specification, with different policy objects supporting different semantics. In general, such policy objects will also have management operations that are different from those defined in this specification.

15.6.4 Access Policies

There are two types of invocation access policies: 1) the Client Invocation Access policy (**Security::SecClientInvocationAccess**) which is used at the client side of an invocation, and 2) the Target Invocation Access policy (**Security::SecTargetInvocationAccess**) which is used at the target side.

There is one policy type for application access. However, no standard administrative interface to this is specified, as different applications have different requirements.

Access Policies control access by *subjects* (possessing Privilege Attributes), to objects, using *rights*. Privilege Attributes have already been discussed (in Section 15.5, “Application Developer’s Interfaces,” on page 15-88); rights are described in the next section.

15.6.4.1 Rights

The standard **Access Policy** objects in a secure CORBA system implement access policy using *rights* (though implementations may define alternative, non-rights-based **Access Policy** objects).

In rights-based systems, **Access Policy** objects *grant* rights to PrivilegeAttributes. For each operation in the interface of a secure object, some set of rights is *required*. Callers must be granted these required rights in order to be allowed to invoke the operation.

Secure CORBA systems provide a **RequiredRights** interface, which allows:

- Object interface developers to express the “access control types” of their operations using standard *rights,* which are likely to be understood by administrators, without requiring administrators to be aware of the detailed semantics of those operations.
- Access-control checking code to retrieve the rights required to invoke an interface’s operations.

A **Required Rights** object is available as an attribute of **Current** in every execution context. Every **Required Rights** object will get and set the same information, so it does not matter which instance of the **RequiredRights** interface is used. The required rights for all operations of all secured interfaces are assumed to be accessible through any instance of **RequiredRights**.

Note that required rights are characteristics of interfaces, *not* of instances. All instances of an interface, therefore, will always have the same required rights.

Note also that because required rights are defined and retrieved through the **RequiredRights** interface, no change to existing object interfaces is required in order to assign required rights to their operations.

Rights Families

This specification provides a standard set of rights for use with the **DomainAccessPolicy** interface defined later in this section. These rights may not satisfy all access control requirements. However; to allow for extensibility, rights are grouped into Rights Families. The **RightsFamily** containing the standard rights is called “**corba**,” and contains four rights: “**g**” (interpreted to mean “**get**”), “**s**” (interpreted to mean “**set**”), “**m**” (interpreted to mean “**manage**”) and “**u**” (interpreted to mean “**use**”). Implementations may define additional Rights Families. **Rights** are always qualified by the **RightsFamily** to which they belong.

15.6.4.2 The SecurityLevel2::RequiredRights Interface

A **Required Rights** object can be thought of as a table (an example Required Rights table appears later in this section). Note that implementations need not manage required rights on an interface-by-interface basis. **Required Rights** objects should be thought of as databases of policy information, in the same way as Interface Repositories are databases of interface information. Thus in many implementations, all calls to the **RequiredRights** interface will be handled by a single Required Rights object instance, or by one of a number of replicated instances of a master Required Rights object instance.

An operation’s entry in the Required Rights table lists a set of rights, qualified (or “tagged”) as usual with the **RightsFamily**. It also specifies a *Rights Combinator*; the rights combinator defines how entries with more than one required right should be interpreted. This specification defines two Rights Combinators: *AllRights* (which means that all rights in the entry must be granted in order for access to be allowed), and *AnyRight* (which means that if any right in the entry is granted, access will be allowed).

Note that the following behaviors of systems conforming to CORBA Security are unspecified and therefore may be implementation-dependent:

- Assignment of initial required rights to newly created interfaces.
- Inheritance of required rights by newly created derived interfaces.

get_required_rights

This operation retrieves the rights required to execute the operation specified by **operation_name** of the interface specified by **obj.** Additionally, **obj**'s interface will be determined and used to retrieve required rights. The returned values are a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry.

```
void get_required_rights(
    in      Object                  obj,
    in      CORBA::Identifier       operation_name,
    in      CORBA::RepositoryId     interface_name,
    out     RightsList              rights,
    out     RightsCombinator        rights_combinator
);
```

Parameters

obj	The object for which required rights are to be returned.
operation_name	The name of the operation for which required rights are to be returned.
interface_name	The name of the interface in which the operation described by **operation_name** is defined, if this is different from the interface of which obj is a direct instance. Not all implementations will require this parameter; consult your implementation documentation.
rights	The returned list of required rights.
rights_combinator	The returned rights combinator.

Return Value

None.

set_required_rights

This operation updates the rights required to execute the operation specified by **operation_name** of the interface specified by **interface_name**. The caller must provide a list of rights and a combinator describing how the list of rights should be interpreted if it contains more than one entry. Note that consistency issues arising from replication of **Required Rights** objects or distribution of the **RequiredRights** interface must be handled correctly by implementations. After a call to **set_required_rights** changes an interface's required rights, all subsequent calls to **get_required_rights**, from any client, must return the updated rights set.

```
void set_required_rights(
    in      string                  operation_name,
    in      CORBA::RepositoryId     interface_name,
    in      RightsList              rights,
    in      RightsCombinator        rights_combinator
);
```

Parameters

operation_name	The name of the operation for which required rights are to be updated.
interface_name	The name of the interface whose required rights are to be updated.
rights	The desired new list of required rights.
rights_combinator	The desired new **RightsCombinator**.

Return Value

None.

15.6.4.3 The SecurityAdmin::AccessPolicy Interface

This is the root interface for the various kinds of invocation access control policy. This interface supports querying of the effective access granted by a set of attributes by an invocation access policy. It inherits the **CORBA::Policy** interface and has a single operation, **get_effective_rights**.

get_effective_rights

This operation returns the current effective rights (of family **RightsFamily**) granted by this **Access Policy** object to the subject possessing all privilege attributes in the list of attributes **attrib_list**.

```
RightsList get_effective_rights(
    in      AttributeList           attrib_list,
    in      ExtensibleFamily        rights_family
);
```

Parameters

attrib_list	A list of attributes obtained from one or more Credentials using the **get_attributes** operation.
rights_family	The family of rights that is desired as a return value.

Return Value

A list of effective rights that are consistent with the **attrib_list** and the access policy, of the family specified by **rights_family**.

Note that this specification does not define how an **Access Policy** object combines rights granted through different Privilege Attribute entries, in case a subject has more than one Privilege Attribute to which the Access Policy grants rights. However, this call will cause the **Access Policy** object to combine rights granted to all privilege attributes in the input **AttributeList** (using whatever operation it has implemented), and return the result of the combination.

Access Decision objects, and applications that check whether access is permitted without using an **Access Decision** object, should use this operation to retrieve rights granted to subjects.

15.6.4.4 Specific Invocation Access Policies

This specification allows different Invocation Access policies to be provided through specialization of the **AccessPolicy** interface.

The provider of each specific Invocation Access policy is responsible for defining its own administrative operations. This specification defines a standard Invocation Access policy interface, including administrative operations, presented in the next section. This standard policy may of course be replaced by, or augmented with, other policies.

15.6.4.5 The Domain AccessPolicy Object

The **Domain Access Policy** object with the **SecurityAdmin::DomainAccessPolicy** interface provides discretionary access policy management semantics. CORBA implementations with policy requirements, which cannot be met by the **Domain Access Policy** abstraction, may choose to implement different **Access Policy** objects. For example, they may choose to implement access control policy management using capabilities.

Domains

This specification defines interfaces for administration of access policy on a domain basis. Each domain may be assigned an access policy, which is applied to all objects in the domain. Each access-controlled object in a CORBA system must be a member of at least one domain.

A **Domain Access Policy** object defines the access policy, which grants a set of named “subjects” (e.g., users), a specified set of “rights” (e.g., **g**, **s**, **m**, **u**) to perform operations on the “objects” in the domain. A Domain Access Policy can be represented by a table whose row labels are the names of subjects, and whose cells are filled with the rights granted to the subject named in that row's label, as in Table 15-2. Note that the use of the Delegation State is discussed in "Delegation State" on page 15-139.

Table 15-2 DomainAccessPolicy

Subject	Delegation State	Granted Rights
alice	initiator	corba:gs--
bob	initiator	corba:g---
cathy	initiator	corba:g---
...		
zeke	initiator	corba:gs--

This **Domain Access Policy** grants the rights "**g**" and "**s**" to Alice and Zeke, and the right "**g**" to Bob and Cathy. The annotation "**corba**" prefixing the granted rights indicates which Rights Family, as defined in the previous section, each of the rights in the table is drawn from. In this case, all rights are drawn from Domain Access Policy's standard "**corba**" Rights Family. The delegation state column is described under "Delegation State" on page 15-139.

Domain Access Policy Use of Privilege Attributes

Administration of principals by individual identity is costly, so the Domain Access Policy aggregates principals for access control. A common aggregation is called a "user group." This specification generalizes the way users are aggregated, using "Privilege Attributes" (as defined in "Access Policies" on page 15-26). Users may have many kinds of privilege attributes, including groups, roles, and clearances (note that user access identities, often referred to simply as "user identities" or "userids," are considered to be a special case of privilege attributes). The **Domain Access Policy** object uses Privilege Attributes as its subject entries.

This specification does not provide an interface for managing user privilege attributes; an implementation of this specification might provide a "User Privilege Attribute Table" enumerating the set of users granted each Privilege attribute. An implementor might provide a user privilege attribute table, shown next in Table 15-3.

Table 15-3 User Privilege Attributes (not defined by this specification)

Users	Privilege Attributes
bob, cathy	group:programmers
zeke	group:administrators

Given the definitions in Table 15-3, we can simplify our Domain Access Policy in Table 15-4 (note that, for convenience, each **PrivilegeAttribute** entry is annotated in Table 15-4 with its **PrivilegeAttribute** type).

Table 15-4 Domain Access Policy (with Privilege Attributes)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	initiator	corba:gs--
group:programmers	initiator	corba:g---
group:administrators	initiator	corba:gs--

Delegation State

The **Domain Access Policy** abstraction allows administrators to grant different rights when a Privilege attribute is used by a delegate than those granted to the same Privilege attribute when used by an initiator (note that "initiator" means the principal issuing the first call in a delegated call chain; that is, the only client in the call chain that is not also a target object). The **Domain Access Policy** shown next illustrates the use of this feature.

Table 15-5 Domain Access Policy (with Delegate Entry)

Privilege Attribute	Delegation State	Granted Rights
access_id:alice	initiator	corba:gs--
access_id:alice	delegate	corba:g---
group:programmers	initiator	corba:g---
group:administrators	initiator	corba:gs--

In Table 15-5, this **Domain Access Policy** grants Alice the "**g**" and "**s**" rights when she accesses an object as an initiator, but only the "**g**" right when a delegate using her identity accesses the same object.

Domain Access Policy Use of Rights and Rights Families

The rights granted to a Privilege Attribute by a **Domain Access Policy** entry must each be "tagged" with the RightsFamily to which they belong. Each **Domain Access Policy** entry can grant its row's **PrivilegeAttribute** rights from any number of different Rights Families.

Implementations may define new Rights Families in addition to the standard "**corba**" family, though this should be done only if absolutely necessary, since new Rights Families complicate the administrator's model of the system.

Access Decision Use of AccessPolicy and RequiredRights

The **Access Decision** object is described in "The Access Decision Object" on page 15-118. It is used at run-time to perform access control checks. **Access Decision** objects rely upon **Access Policy** objects to provide the policy information upon which their decisions are based.

To complete the example, imagine that we have the set of object instances shown in Table 15-6..

Table 15-6 Interface Instances

Objects	**Interface**
obj_1, obj_8, obj_n	c1
obj_2, obj_5	c2
obj_12	c3

The **Domain Access Policy** object illustrated in Table 15-7 has been updated to include a list of rights of type "other" granted to each of the Privilege attributes..

Table 15-7 Domain Access Policy (with Required Rights Mapping)

Privilege Attribute	**Delegation State**	**Granted Rights**
access_id:alice	initiator	corba: gs-- other: -u-m-s
access_id:alice	delegate	corba: g--- other: ------
group:programmers	initiator	corba: g--- other: -u----
group:administrators	initiator	corba: gs-- other: ------

Table 15-8 shows Required Rights for three object interfaces (c1, c2, and c3), using the standard Rights Family "**corba**" and a second Rights Family, "other," whose rights set is assumed to be {g, u, o, m, t, s}.

Table 15-8 Required Rights for Interfaces c1, c2, and c3

Required Rights	**Rights Combinator**	**Operation**	**Interface**
corba:s	all	m1	c1
corba:gs	any	m2	
other:u	all	m3	c2

Table 15-8 Required Rights for Interfaces c1, c2, and c3

Required Rights	Rights Combinator	Operation	Interface
other:ms	all	m4	
other: s	all	m5	c3
corba:gs	all	m6	

Using this, we can calculate the effective access granted by this Domain Access Policy.

- alice can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but may execute only m2 as a delegate.
- alice can execute operations m3 and m4 of objects obj_2, and obj_5 as an initiator, but may execute no operations of obj_2 and obj_5 as a delegate.
- alice can execute operations m5 and m6 of object obj_12 as an initiator, but may execute no operations as a delegate.
- "programmers" can execute operation m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- "programmers" can execute operation m3 of objects obj_2 and obj_5 as an initiator, but no operations as a delegate.
- "administrators" can execute operations m1 and m2 of objects obj_1, obj_8, and obj_n as an initiator, but no operations as a delegate.
- "administrators" can execute operations m5 and m6 of object obj_12 as an initiator, but no operations as a delegate.

15.6.4.6 The SecurityAdmin::DomainAccessPolicy Interface

The **Domain Access Policy** object provides operations for managing access policy through the **DomainAccessPolicy** interface.

Each domain manager may have at most one **Access Policy** object, and therefore at most one **Domain Access Policy** (though an object instance may have more than one domain manager, and therefore, more than one **Domain Access Policy**). The **DomainAccessPolicy** interface inherits the **AccessPolicy** interface and defines operations to specify which subjects can have which rights as follows.

grant_rights

This operation grants the specified **rights** to the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to grant rights to a single privilege attribute.

```
void grant_rights(
    in      Attribute                          priv_attr,
```

```
    in      DelegationState                 del_state,
    in      ExtensibleFamily                rights_family,
    in      RightsList                      rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights_family	The family of rights to be affected.
rights	The list of rights to be granted.

Return Value

None.

revoke_rights

Revokes the specified **rights** of the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to revoke rights granted to a single privilege attribute.

```
void revoke_rights(
    in      Attribute                       priv_attr,
    in      DelegationState                 del_state,
    in      ExtensibleFamily                rights_family,
    in      RightsList                      rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights_family	The family of rights to be affected.
rights	The list of rights to be revoked.

Return Value

None.

replace_rights

Replaces the current rights of the privilege attribute **priv_attr** in delegation state **del_state** with the **rights** provided as input.

Utilities that manage access policy should use this operation to replace rights granted to a single privilege attribute in cases where using **grant_rights** and **revoke_rights** is inappropriate. For example, **replace_rights** might be used to change an **access_id**'s authorizations to reflect a change in job description (since the change in authorization in this case is related to the duties of the new job rather than to the current authorizations granted to the user owning the **access_id**).

```
void replace_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      ExtensibleFamily        rights_family,
    in      RightsList              rights
);
```

Parameters

priv_attr	Privilege attributes to be affected.
del_state	Delegation state to be set.
rights_family	The family of rights to be affected.
rights	The list of rights to be replaced.

Return Value

None.

get_rights

Returns the current rights (of type **RightsList**) of the privilege attribute **priv_attr** in delegation state **del_state**.

Utilities that manage access policy should use this operation to retrieve rights granted to an individual privilege attribute.

```
RightsList get_rights(
    in      Attribute               priv_attr,
    in      DelegationState         del_state,
    in      ExtensibleFamily        rights_family
);
```

Parameters

priv_attr	Privilege attributes to which the requested rights are granted.
del_state	Delegation state to be set.
rights_family	The family of rights to be affected.

Return Value

A list of rights granted to the specified privilege attribute of the specified rights family in the specified delegation state.

15.6.5 Audit Policies

There are two invocation audit policies: 1) the **SecClientInvocationAudit** policy, which is used at the client side of an invocation, and 2) the **SecTargetInvocationAudit** policy, which is used at the target side. There is also an application audit policy type.

Audit policy administration interfaces are used to specify the circumstances under which object invocations and application activities in this domain are audited. As for access policies, this specification allows different audit policies to be specified, which may have different administrative interfaces.

Different audit policies are potentially possible, which allow a great range of options of what to audit. Some of these are needed to respond to the problem of getting the useful information, without generating huge quantities of audit information.

Examples of what events could be audited during invocation include:

- Specified operations on objects.
- Failed operations (i.e., those that raise an exception) on specified object types in a domain.
- Use of certain operations during certain time intervals (e.g., overnight).
- Access control failures on specified operations.
- Operations done by a specified principal.
- Combinations of these.

Note that many of these events may be related to the business application. For example, an operation of *update_bank_account* is a business, rather than system, operation. However, some events are mainly of interest to a Privilege administrator (e.g., access failures to systems objects).

Application audit policies may audit similar types of events, though these are often related to application functions, not object invocations.

15.6.5.1 The SecurityAdmin::AuditPolicy Interface

The **AuditPolicy** interface can be used to administer both client and target invocation audit policies.

This standard audit policy is used to specify, for a set of event families and event types, the selectors to be used to define which events are to be audited.

These are related to the selectors used in **audit_needed** (of **Audit Decision** object, interface **AuditDecision**) and **audit_write** (of **Audit Channel** object, interface **AuditChannel**) as follows in Table 15-9.

Table 15-9 Standard Audit Policy

Selector Type	**Value on audit_needed and audit_write**	**Value Administered**
InterfaceRef	from object reference	object type
ObjectRef	object reference	none - the policy applies to all objects in the domain
Operation	op_name	operation
Initiator	credential list	security attributes (audit_id and privileges)
Success Failure	boolean	boolean
Time	utc when event occurred	time interval during which auditing is needed

Note that audit policy is managed on an audit policy domain basis. Assignment of initial audit selectors to newly created domains is unspecified and hence may be implementation-dependent.

The following operations are available on the Audit Policy object.

set_audit_selectors

This operation defines the selectors to be used to decide whether to audit the specified event families and types.

```
void set_audit_selectors(
    in      CORBA::InterfaceDef     object_type,
    in      AuditEventTypeList      events,
    in      SelectorValueList       selectors
);
```

Parameters

object_type	The type of objects for which an audit policy is being set. If this is nil, all object types are implied.
events	Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.
selectors	The values of the selectors to be used.

Return Value

None.

clear_audit_selectors

Clears all audit selectors for the specified event families and types.

```
void clear_audit_selectors(
    in      CORBA::InterfaceDef      object_type,
    in      AuditEventTypeList       events,
);
```

Parameters

object_type	The type of objects for which an audit policy is being cleared. If this is nil, all object types are implied.
events	Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.

Return Value

None.

replace_audit_selectors

Replaces the specified selectors.

```
void replace_audit_selectors(
    in      CORBA::InterfaceDef      object_type,
    in      AuditEventTypeList       events,
    in      SelectorValueList        selectors
);
```

Parameters

object_type	The type of objects for which an audit policy is being replaced. If this is nil, all object types are implied.
events	Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.
selectors	The values of the selectors to be used.

Return Value

None.

get_audit_selectors

Obtains the current values of the selectors for the specified event family or event.

```
SelectorValueList get_audit_selectors(
    in      CORBA::InterfaceDef     object_type,
    in      AuditEventTypeList      events,
);
```

Parameters

object_type	The type of objects for which an audit policy is being obtained. If this is nil, all object types are implied.
events	Event types are specified as family and type ids. If the type id is zero (**AuditAll**), the selectors apply to all event types in that family.

Return Value

A list of selector values.

set_audit_channel

Specifies the identity of the audit channel to be used with this audit policy. The actual audit channel object corresponding to this id is provided to the user by the corresponding Audit Decision object.

```
void set_audit_channel(
    in      AuditChannelId    audit_channel_id
);
```

Parameters

audit_channel_id	A unique identifier associated with an audit channel.

Return Value

None.

15.6.6 Secure Invocation and Delegation Policies

These policies affect the way secure communications between client and target are set up, and then used. There are three policies here:

1. **Security::SecClientSecureInvocation** policy, which specifies the client policy in terms of trust in the target's identity and protection requirements of the communications between them.

2. **Security::SecTargetSecureInvocation** policy, which specifies the target policy in terms of trust in the client's identity and protection requirements of the communications between them.

3. **Security::SecDelegation** policy, which specifies whether credentials are delegated for use by the target when a security association is established between client and target. This is a client-side policy.

In all these cases, there is a standard policy interface for administering the policy options. Unlike access and audit policies, this is not replaceable. The standard policy administration operations allow support of a range of policies.

15.6.6.1 Secure Invocation Policies

These are used to set client and target invocation policies which specify both a set of required secure association options and a set of supported options that control how:

- The security association is made, for example, whether trust between client and target is established (implying authentication if the client and target are not in the same identity domain).
- Messages using that association are protected, for example, the levels of integrity and confidentiality.

The administrator should specify the required association options, but will often not need to specify the supported options as these default to the ones supported by the security mechanism used. However, the administrator could choose to restrict what is supported, and in this case, should specify supported options.

Some implementations may support separate sets of association options for communications in the request direction and the reply direction (e.g., for an application that requires no protection on the request, but confidentiality on the reply). Conforming implementations are not required to support this unidirectional feature. Some selectable policy options may not be meaningful to set for a certain direction (e.g., the **EstablishTrustInTarget** option is not meaningful for a reply).

Both **SecClientSecureInvocation** and **SecTargetSecureInvocation** type policy objects support the same interface, though not all of the selectable policy options are meaningful to both client and target.

Required and Supported Secure Invocation Policy

For both the **SecClientSecureInvocation** and **SecTargetSecureInvocation** policies, a separate set of secure association options may be established to indicate required policy and supported policy. The required policy indicates the options that an object requires for communications with a peer. The supported policy specifies the options that an object can support if requested by a communicating peer.

The required options indicate the minimum requirements of the object, stronger protection is not precluded.

15.6.6.2 Secure Association Options

The selectable secure association options (**Security::AssociationOptions**) are listed next with a description of their semantics for required policy and supported policy.

NoProtection

- Required semantics: the object's minimal protection requirement is unprotected invocations.
- Supported semantics: the object supports unprotected invocations.

Integrity

- Required semantics: the object requires at least integrity-protected invocations.
- Supported semantics: the object supports integrity-protected invocations.

Confidentiality

- Required semantics: the object requires at least confidentiality-protected invocations.
- Supported semantics: the object supports confidentiality-protected invocations.

DetectReplay

- Required semantics: the object requires replay detection on invocation messages.
- Supported semantics: the object supports replay detection on invocation messages.

DetectMisordering

- Required semantics: the object requires sequence error detection on fragments of invocation messages.
- Supported semantics: the object supports sequence error detection on fragments of invocation messages.

EstablishTrustInTarget

- Required semantics: On client policy, the client requires the target to authenticate its identity to the client. On target policy, this option is not meaningful.
- Supported semantics: On client policy, the client supports having the target authenticate its identity to the client. On target policy, the target is prepared to authenticate its identity to the client.

EstablishTrustInClient

- Required semantics: On client policy, this option is not meaningful. On target policy, the target requires the client to authenticate its privileges to the target.
- Supported semantics: On client policy, the client is prepared to authenticate its privileges to the target. On target policy, the target supports having the client authenticate its privileges to the target.

Note – On an invocation, if both the client and target policies specify that peer trust is needed, mutual authentication of client and target is generally required.

If the target accepts unauthenticated users as well as authenticated ones, the **EstablishTrustInClient** option may be set for supported policy, but not for required policy. This allows unauthenticated clients to use this target (subject to access controls); the target can still insist on only authenticated users for certain operations by using access controls.

15.6.6.3 *The SecurityAdmin::SecureInvocationPolicy Interface*

The **SecureInvocationPolicy** interface provides the following operations:

set_association_options

This operation of the **SecurityAdmin::SecureInvocationPolicy** interface (PolicyType **SecClientSecureInvocation** and **SecTargetSecureInvocation**) is used to set the secure association options for objects in the domain to which the policy applies. Separate options may be set for particular object types by using the **object_type** parameter.

This call allows requesting a different set of association options for communication in the request direction versus the reply direction, although conforming implementations are not required to support this feature. The "**request**" and "**reply**" options sets are treated as overrides to the "**both**" options set when evaluating policy for a single communication direction. Implementations should raise the CORBA::BAD_PARAM exception if an unsupported direction is requested on this call.

Not all selectable association options are meaningful for every policy set. For example, **EstablishTrustInClient**, which is meaningful for the **SecTargetSecureInvocation** policy, is not meaningful as a requirement for the **SecClientSecureInvocation** policy. Likewise, certain association options do not make sense when applied to only a single direction (e.g., **EstablishTrustInTarget** is not meaningful for communication in the reply direction). An implementation may choose whether to raise an exception or silently ignore requests for invalid association options.

```
void set_association_options(in CORBA::InterfaceDefobject_type,
    in      RequiresSupports            requires_supports,
    in      CommunicationDirection      direction,
    in      AssociationOptions          options
);
```

Parameters

object_type	The type of objects that the association options apply to. If this is nil, all object types are implied
requires_supports	Indicates whether the operation applies to the required options or the supported options
direction	Indicates whether the options apply to only the request, only the reply, or to both directions of the invocation.
options	Indicates requested secure association options by setting the corresponding options flags.

Return Value

None.

get_association_options

This is used to find what secure association options apply on **SecClientSecureInvocation** and **SecTargetSecureInvocation** policy objects for the required or supported policy, for the indicated direction, and for the specified object type.

Implementations should raise the CORBA::BAD_PARAM exception if an unsupported direction is requested on this call.

```
AssociationOptions get_association_options(
    in      CORBA::InterfaceDef          object_type,
    in      RequiresSupports             requires_supports,
    in      CommunicationDirection       direction
);
```

Parameters

object_type	The type of objects that the association options apply to. If this is nil, all object types are implied
requires_supports	Indicates whether the operation applies to the required options or the supported options
direction	Indicates whether the options apply to only the request, only the reply, or to both directions of the invocation.

Return Values

The association options flags set for this policy.

15.6.6.4 The SecurityAdmin::DelegationPolicy Interface

The **Delegation Policy** object, which has the **SecurityAdmin::DelegationPolicy** interface, controls which credentials are used when an intermediate object in a chain invokes another object.

set_delegation_mode

The **set_delegation_mode** operation specifies which credentials are delegated by default at an intermediate object in a chain where objects invoke other objects. This default can be overridden by the object at run time.

```
void set_delegation_mode(
    in      CORBA::InterfaceDef     object_type,
    in      DelegationMode          mode
);
```

Parameters

object_type	The type of objects to which this delegation policy applies.	
mode	The delegation mode. Options are:	
	SecDelModeNo Delegation	The intermediate's own credentials are used for future invocations.
	SecDelModeSimple Delegation	The initiating principal credentials are delegated.
	SecDelModeComposite Delegation	Both the received credentials and the intermediate object's own credentials are passed (if the underlying security mechanism supports this). The requester's credentials and the intermediate's own credentials may be combined into a single credential, or kept separate, depending on the underlying security mechanism.

Return Value

None.

get_delegation_mode

Returns the delegation mode associated with the object.

```
DelegationMode get_delegation_mode(
    in      CORBA::InterfaceDef     object_type
);
```

Parameters

object_type	The type of object for which delegation mode is requested.

Return Value

The delegation mode of the object type specified by the **object_type** parameter.

15.6.7 Non-repudiation Policy Management

This section defines interfaces for management of non-repudiation policy.

Non-repudiation policies define the following:

- Rules for the generation of evidence, such as the trusted third parties which may be involved in evidence generation, the roles in which they may be involved and the duration for which the generated evidence is valid.
- Rules for the verification of evidence, for example, the interval during which a trusted third party may legitimately declare its key to have been compromised or revoked.
- Rules for adjudication, for example, which authorities may be used to adjudicate disputes.

The non-repudiation policy itself may be used by the adjudicator when resolving a dispute. For example, the adjudicator might refer to the non-repudiation policy to determine whether the rules for generation of evidence have been complied with.

For each type of evidence, a policy defines a validity duration and whether trusted time must be used to generate the evidence.

For each non-repudiation mechanism, a policy defines the set of trusted third parties ("authorities"), which may be used by the mechanism. A policy also defines, for each mechanism, the maximum allowable "skew" between the time of generation of evidence and the time of countersignature by a trusted time service; if the interval between these two times is larger than the maximum skew, the time is not considered to be trusted.

For each authority, a policy defines which roles the authority may assume, and a time offset, relative to evidence generation time, which allows computation of the last time at which the authority can legitimately declare its key to have been compromised or revoked. For example, if an authority has a defined **last_revocation_check_offset** of negative one hour, then all revocations taking effect earlier than one hour before the generation of a piece of evidence will render that evidence invalid; no revocation taking place later than one hour before the generation of the evidence will affect the evidence's validity. Note that the **last_revocation_check_offset** is inclusive, in the sense that all revocations occurring up to and including the time defined by **generation_time + offset** are considered effective.

15.6.7.1 Data Types for Non-repudiation Policy Management Interfaces

The following data types are used by the NR policy management operations.

```
module NRservice {

    struct EvidenceDescriptor {
        EvidenceType            evidence_type,
        DurationInMinutes       evidence_validity_duration,
        boolean                 must_use_trusted_time,
    };

    typedef sequence <EvidenceDescriptor> EvidenceDescriptorList;

        struct AuthorityDescriptor {
            string                          authority_name,
            string                          authority_role,
            TimeOffsetInMinutes last_revocation_check_offset
            // may be >0 or <0; add this to evid. gen. time to
            // get latest time at which mech. will check to see
            // if this authority's key has been revoked.
};

typedef sequence <AuthorityDescriptor> AuthorityDescriptorList;

    struct MechanismDescriptor {
        NRMech                  mech_type,
        AuthorityDescriptorList authority_list,
        TimeOffsetInMinutes         max_time_skew
        // max permissible difference between evid. gen. time
        // and time of time service countersignature
        // ignored if trusted time not reqd.
    };

    typedef sequence <MechanismDescriptor> MechanismDescriptorList;
};
```

15.6.7.2 The NRservice::NRPolicy Interface

The **NRPolicy** interface has the **get_NR_policy_info** and **set_NR_policy_info** operations, and like all other **Policy** interfaces it derives from the **CORBA::Policy** interface.

get_NR_policy_info

Returns information from a non-repudiation policy object.

```
void get_NR_policy_info(
    out     ExtensibleFamily                NR_policy_id,
    out     unsigned long                   policy_version,
```

```
    out     TimeT                       policy_effective_time,
    out     TimeT                       policy_expiry_time,
    out     EvidenceDescriptorList      supported_evidence_types,
    out     MechanismDescriptorList     supported_mechanisms
);
```

Parameters

NR_policy_id	The identifier of this non-repudiation policy.
policy_version	The version number of this non-repudiation policy.
policy_effective_time	The time at which this policy came into effect.
policy_expiry_time	The time at which this policy expires.
supported_evidence_types	The types of evidence that can be generated under this policy.
supported_mechanisms	The non-repudiation mechanisms which can be used to generate and verify evidence under this policy.

Return Value

None.

set_NR_policy_info

Updates non-repudiation policy information.

```
boolean set_NR_policy_info(
    in      MechanismDesciptorList      requested_mechanisms,
    out     MechanismDescriptorList     actual_mechanisms
);
```

Parameters

requested_mechanisms	The non-repudiation mechanisms to be supported under this policy.
actual_mechanisms	The non-repudiation mechanisms now supported under this policy.

Return Value

TRUE	The requested mechanisms were all set.
FALSE	If the actual mechanisms returned differ from those requested.

15.7 Implementor's Security Interfaces

This section addresses Security Service replaceability. This section defines the security service interfaces that allow different security service implementations to be substituted, whether or not the generic ORB service interfaces are supported (see Section 15.7.2, "Implementation-Level Security Object Interfaces," on page 15-161, for details).

Appendix D, "Guidelines for a Trustworthy System" on page 15-329, offers additional guidance to implementors of secure ORBs, including a discussion of using protection boundaries to separate components, depending on the level of security required.

The description of security interceptors in Section 15.7.1, "Security Interceptors," on page 15-156 (particularly that in Invocation Time Policies), specifies how client- and target-side policies and client preferences are used to decide what policy options to enforce. This definition of how the options are used applies whether the ORB conforms to the replaceability options or not. The interceptor facility that this is based on is defined in the Interceptors chapter of the *Common Object Request Broker: Architecture and Specification*.

None of the interfaces defined in this section affect the application and administrator's views described in Section 15.5, "Application Developer's Interfaces," on page 15-88, and Section 15.6, "Administrator's Interfaces," on page 15-130.

15.7.1 Security Interceptors

This section describes the interceptors that can be used for implementing the security services.

The ORB Services replaceability package requires implementation of two security interceptors (see the Interceptors chapter of the *Common Object Request Broker: Architecture and Specification*):

- **Secure Invocation Interceptor**. This is a message-level interceptor. At bind time, this establishes the security context required to support message protection. When processing a request, it is a message-level interceptor that uses cryptographic services to provide message protection and verification. It is able to check and protect messages (requests and replies) for both integrity and confidentiality.
- **Access Control Interceptor.** This is a request-level interceptor, which determines whether an invocation should be permitted. This interceptor also handles auditing of general invocation failures, but not related to denial of access (access-control denial failures are audited within the **Access Decision** object, which is called by this interceptor to check access control).

This specification does not define a separate audit interceptor, as the other interceptors' implementations or the security service implementations call Audit Service interfaces directly if the events for which they are responsible are to be audited.

The security interceptors implement security functionality by calling the replaceable security service objects (defined later in this section) as shown in Figure 15-54.

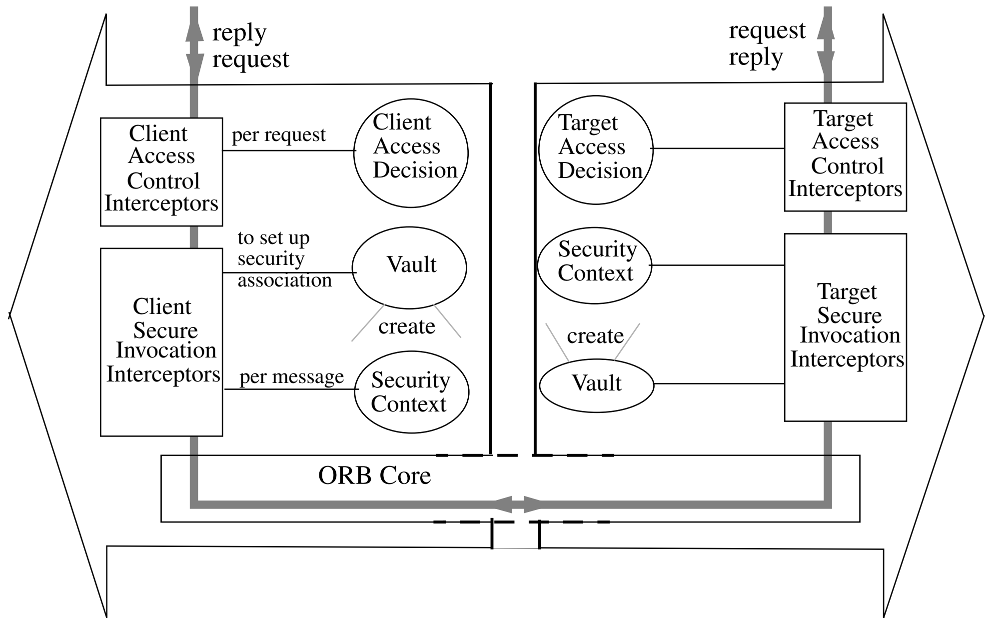


Figure 15-54 Security Functionality Implemented by Security Service Objects

The diagram shows the order in which security interceptors are called. Other interceptors may also be used during the invocation. The order in which other interceptors are called in relationship to security interceptors depends on the type of interceptor.

At the client:

- In general, the access control interceptor should be called first (to avoid unnecessary processing of the request by other interceptors when permission to perform the request is denied).
- All request-level interceptors (e.g., transaction or replication ones) are called before the secure invocation interceptor, as the secure invocation interceptor is a message-level interceptor.

  The secure invocation interceptor should ordinarily be the last interceptor invoked (because message protection may encrypt the request so that the code implementing a further interceptor will not understand it). Even if only integrity protection is used, the integrity check will fail if the message has been altered in any way. Note that data compression and data fragmentation should be applied before the message-protection interceptor is called.

At the target, analogous rules apply to the interceptors in the reverse order.

15.7.1.1 Invocation Time Policies

Interceptors decide what security policies to enforce on an invocation as follows:

- They call the **SecurityLevel2::Current::get_policy** operation defined in Section 15.5, "Application Developer's Interfaces," on page 15-88, to find what policies apply to this client (at the client side) or this target (at the target side).
- At the client side, the security hints in the target object reference are used to find what policies apply to the target object and what security mechanisms and protocols are supported. This uses operations on the object reference.
- At the client, the overrides set by the client on the credentials or target object reference and the security supported by the mechanism in the client's environment are taken into account. The Secure Invocation interceptor uses **SecurityLevel2::Current::get_credentials** and **Object::get_policy**.

The **Current::get_policy** operation may be used to get any of the following policies:

- The invocation access policies of the current execution context. These are used by the access control interceptor to check whether access is permitted.
- The invocation audit policy. This is used by interceptors and security services to check whether to audit events during an invocation.
- The secure invocation policy. This is used by the secure invocation interceptor at bind time. It uses **SecureInvocationPolicy::get_association_options** as defined in Section 15.6, "Administrator's Interfaces. The secure invocation policies (and hints in the object reference) specify required and supported values. The interceptor checks that the required values can be supported, and will not continue with the invocation if the client's requirements are not met. If the target's requirements are not met, the invocation may optionally proceed, allowing policy enforcement at the target.
- The invocation delegation policy. This is used by the secure invocation interceptor at bind time. The interceptor calls **SecureInvocationPolicy::get_delegation_mode** to retrieve this information.

15.7.1.2 Secure Invocation Interceptor

At bind time, the secure invocation interceptor establishes a security context, which the client initiating the binding can use to securely invoke the target object designated by the object reference used in establishing the binding. At object invocation time, the secure invocation interceptor is called to use the (previously established) security context to protect the message data transmitted from the client to the invoked target object.

Please note that the remainder of this section assumes that security interceptors are implemented using the security services replaceability interfaces defined in this specification; interceptors built for implementations which do not provide the security services replaceability interfaces will have similar responsibilities, but will obviously make different calls.

It should also be noted that binding takes place implicitly and the exact point at which it occurs can vary from one ORB to another. All that one can be certain of is that a binding exists when an invocation of an operation takes place. There is no certainty that the same binding will be used in subsequent invocations. Consequently, the discussion that follows is about binding states and what must happen when the act of implicit binding is executed by the ORB. All reference to the term "Bind" should be interpreted as such.

Bind Time - Client Side

The Secure Invocation interceptor's client bind time functions are used to:

- Find what security policies apply.
- Establish a security association between client and target. This is done on first invoking the object, but may be repeated when changes to the security context occur.

Security policies relevant to this interceptor are the client-secure invocation and delegation policies. To retrieve the invocation policy objects, the Secure Invocation interceptor calls the **get_policy** operation.

The interceptor checks if there is already a suitable security context object for this client's use of this target. If a suitable context already exists, it is used. If no suitable context exists, the interceptor establishes a security association between the client and target object (see "Establishing Security Associations" on page 15-22).

The client interceptor calls **Vault::init_security_context** to request the security features (such as QOP, delegation) required by the client policy, client overrides, and target (as defined in its object reference). The **Vault** returns a security token to be sent to the target, and indicates whether a continuation of the exchange is needed. It also returns a reference to the newly-created **Security Context** object for this client-target security association. (The way trust is established depends on policy, the security technology used, and whether both client and target object are in the same identity domain. It may involve mutual authentication between the objects and negotiation of mechanisms and/or algorithms.)

The interceptor constructs the association establishment message (including the security token, which must be transferred to the target to permit it to establish the target-side **Security Context** object). The association establishment message may be constructed in one of two ways:

1. When only the initial security token is needed to establish the association, the association establishment message may also include the object invocation in the buffer (i.e., the request) supplied to the interceptor when it was invoked by **send_message**. After constructing the association establishment message, the interceptor invokes **send,** which results in the ORB sending the message to the target. After receipt at the target, the association establishment message is intercepted by the Secure Invocation Interceptor in the target, which at bind time calls **Vault::accept_security_context** to create the target **Security Context** object (if needed).

2. When several exchanges are required to establish the security association, the association establishment message is sent separately, in a message that does not include the object invocation in the buffer (i.e., the request), again using **send**. This message is intercepted in the target and the **Vault** called to create the **Security Context** object. However, in this case, the target interceptor must generate another security token and send it back to the client interceptor. The client interceptor calls the **Security Context** object with a **continue_security_context** operation passing the token returned from the target to check if trust has now been established. There may be several exchanges of security tokens to complete this. Once the security association has been established, the original client object invocation (i.e., request) is sent in a separate association establishment message.

Details of the transformation to the request and the association establishment message formats appear in Section 15.8, "Security Interoperability Protocols," on page 15-172.

Bind Time - Target Side

The secure invocation interceptor's target bind functions:

- Find the target secure invocation policies.
- Respond to association establishment messages from the client to establish security associations.

On receiving an association establishment message, the target secure invocation interceptor separates it (if needed) into the security token and the request message and uses the **Vault** (if there is no security context object yet) or the appropriate **Security Context** object to process the security token. As previously described, this may result in exchanges with the client. Once the association is established, the message protection function described next is used to reclaim the request message and protect the reply.

Message Protection (Client and Target Sides)

The Secure Invocation Interceptor is used after bind time for message protection, providing integrity and/or confidentiality protection of requests and responses, according to quality of protection requirements specified for this security association in the active **Security Context** object.

The Secure Invocation Interceptor's **send_message** method calls **SecurityContext::protect_message,** and its **receive_message** method calls **SecurityContext::reclaim_message**, in each case using the appropriate **Security Context** object.

15.7.1.3 Access Control Interceptor

Bind Time

At bind time, the client access control interceptor uses **Current::get_policy** to get the **SecClientInvocationAccess** policy and **SecClientInvocationAudit** policy. The target access control interceptor uses the **get_policy** interface on the Current object to get the **SecTargetInvocationAccessPolicy** and **SecTargetInvocationAudit** policy.

Access Decision Time

The Access Control Interceptor decides whether a request should be allowed or disallowed.

Access control decisions may be made at the client side, depending on the client access control policy, and at the target-side depending on the target's access control policy. Target-side access controls are the norm; client-side access controls can be used to reduce needless network traffic in distributed ORBs. Note that in some ORBs, system integrity considerations may make exclusive reliance on client-side access control enforcement undesirable.

The Access Control Interceptor **client_invoke** and **target_invoke** methods invoke the **access_allowed** method of the **Access Decision** object, specifying the appropriate authorization data. The access decision returns a boolean specifying whether the request should be allowed or disallowed.

The Access Control Interceptor does not know what sort of policy this **Access Decision** object supports. It may be ACL-based, capability-based, label-based, etc. It also does not know if the **Access Decision** object uses the credentials exactly as passed, or takes the identity from the credentials and uses these to find further valid privileges if needed for this principal from a trusted source.

The Access Control Interceptor may also check if this invocation attempt should be audited, by calling the **audit_needed** operation on the **Audit Decision** object; if this call indicates that the invocation attempt should be audited, the Access Control Interceptor uses the **AuditChannel** interface to write the appropriate audit record.

This interceptor does not transform the request. It either passes the request unchanged to continue processing the request, or it aborts the request by returning with an appropriate exception (e.g., CORBA::NO_PERMISSION if **AccessDecision::access_allowed** returns False).

15.7.2 Implementation-Level Security Object Interfaces

The interfaces described in this section are all provided by the underlying security infrastructure and the Object Security Service is a client of these interfaces. Since the interfaces are internal to the ORB Security implementation, all these interfaces are **locality-constrained**.

This specification defines the following implementation-level security object interfaces to support security service replaceability:

- **Vault** is used to create a security context for a client/target-object association.
- **Security Context** objects hold security information about the client-target security association and are used to protect messages.
- **Credentials** object is used for passing **Credentials** information between the security infrastructure and the ORB Security Services.
- **Access Decision** objects are used (usually by Access Control Interceptors) to decide if requests should be allowed or disallowed.
- **Audit Decision.** objects are used to decide if events are to be audited.
- **Audit Channel** objects are used to write audit records to the audit trail.
- **Principal Authenticator** object is used for authenticating a principal.
- **NRCredentials** object is used for passing non-repudiation credentials informations.

While many of these objects have interfaces that are defined in the context of user interfaces, the Security Replaceability versions of these objects are implemented by the underlying security infrastructure, and the ORB Security Services are the clients of these interfaces.

15.7.2.1 The Vault Object

The Vault object with the **SecurityReplaceable::Vault** interface facilitates establishing security contexts between clients and targets when these are in different trust domains, so that authentication is required to establish trust. It is a **locality-constrained** object. Implementations of the Vault are responsible for calling **AuditDecision::audit_needed** to determine whether the audit policy requires auditing of successful and/or failed access control checks, and for calling **AuditChannel::audit_write** whenever audit is needed.

15.7.2.2 The SecurityReplaceable::Vault Interface

The **Vault** operations are described below. Note that if an invocation of a **Vault** operation results in an incomplete **Security Context** (i.e., one which requires continued dialogue to complete), the continuation of the dialogue is accomplished using the interface of the incomplete **Security Context** object rather than the **Vault** interface.

init_security_context

This is used by the association interceptor (or the ORB if separate interceptors are not implemented) at the client to initiate the establishment of a security association with the target. As part of this, it creates the Security Context object, which will represent the client's view of the shared security context.

```
AssociationStatus init_security_context(
    in      CredentialsList             creds_list,
    in      SecurityName                target_security_name,
    in      Object                      target,
    in      DelegationMode              delegation_mode,
```

```
    in      OptionsDirectionPairList     association_options,
    in      MechanismType                mechanism,
    in      Opaque                       mech_data,
    in      Opaque                       chan_bindings,
    out     Opaque                       security_token,
    out     SecurityContext              security_context
);
```

Parameters

creds_list	The credentials to be used to establish the security association. There is normally only one credential object: either the default ones from Current, or the ones specified in an override operation on the target object reference. However, for composite, combined or traced delegation, more than one credential object is needed.
target_security_name	The security name of the target as set in its object reference.
target	The target object reference.
delegation_mode	The mode of delegation to employ. The value is obtained by combining client policy and application preferences as described in Invocation Time Policies under Section 15.7.1, "Security Interceptors," on page 15-156.
association_options	A sequence of one or more pairs of secure association options and direction. The options include such things as required peer trust and message protection. Normally, one pair will be specified, for the "both" direction. Implementations that support separate association options for requests and replies may supply an additional options set for each direction supported. These values are obtained from a combination of the client's security policy, the hints in the target object reference, and any requests made by the application.
mechanism	Normally NULL, meaning use default mechanism for security associations. Otherwise, it contains the security mechanism(s) requested. (These may have been obtained from the target object reference.)
mech_data	Any data specific to the chosen mechanism, as found in the target object reference

chan_binding	Normally NULL (zero length). If present, they are channel bindings as in GSS-API.
security_token	The token to be transmitted to the target to establish the security association. Note that this may take several exchanges, but operations required at the client to continue the establishment of the association are on the **Security Context** object.
mech_data	Any data specific to the chosen mechanism, as found in the target object reference

Return Value

The return value is used to specify the result of the operation.

SecAssocSuccess	Indicates that the security context has been successfully created and that no further interactions with it are needed to establish the security association.
SecAssocFailure	Indicates that there was some error, which prevents establishment of the association.
SecAssocContinue	Indicates that the association procedure needs more exchanges.

accept_security_context

This is used by the association interceptor (or ORB) at the target to accept a request from the client to establish a security association. As part of this, it creates the **Security Context** object, which will represent the target's view of the shared security context.

```
AssociationStatus accept_security_context(
    in      CredentialsList         creds_list,
    in      Opaque                  chan_bindings,
    in      Opaque                  in_token,
    out     Opaque                  out_token,
    out     SecurityContext         security_context
);
```

Parameters

creds_list	The credentials of the target. Note that this may be the credentials of the trust domain, not the individual object.
chan_bindings	If present, the channel bindings are as in GSS-API.

in_token	The security token transmitted from the client.
out_token	If establishment of the security association is not yet complete, this contains the security token to be transmitted to the client to continue the security dialogue. Note that any further operations needed to complete the security association are on the security context object.
security_context	The **Security Context** object at the target which represents the shared security context between client and target.

Return Value

SecAssocSuccess	Indicates that the security context has been successfully created and no further interactions with it are needed to establish the security association.
SecAssocFailure	Indicates that there was some error that prevents establishment of the association.
SecAssocContinue	The first stage of establishing the security association has been successful, but it is not complete. The **out_token** contains the token to be returned to continue it.

get_supported_mechs

Returns the mechanism types supported by this Vault object and the association options these support.

```
MechandOptionsList get_supported_mechs ();
```

Parameters

None.

Return Value

The list of mechanism types supported by this Vault object and the association options they support.

15.7.2.3 The Security Context Object

A **Security Context** object with the **SecurityReplaceable::SecurityContext** interface represents the shared security context between a client and a target. It is a **locality-constrained** object. It is used as follows:

- By the security association interceptors to complete the establishment of a security association between client and target after the **Vault** has initiated this.

- By the message protection interceptors in protecting messages for integrity and/or confidentiality.
- In response to a target object's request to **Current** for privileges and other information (sent from the client) about the initiating principal.
- In response to a target object's request to **Current** to supply one (or more) credentials object(s) from incoming information about principal(s).
- To check if the security context is valid, and if not, try and refresh it.

15.7.2.4 *The SecurityReplaceable::SecurityContext Interface*

The **SecurityReplaceable::SecurityContext** interface has the following attributes and operations:

received_credentials

The **received_credentials** readonly attribute returns a list of **Credentials** that were received from the invoker.

```
readonly attribute CredentialsList received_credentials;
```

Return Value

List of object references to receive credentials objects.

security_features

The **security_features** readonly attribute returns a list of security features that are available in the SecurityContext.

```
readonly attribute SecurityFeatureValueList security_features;
```

Return Value

List security feature values.

continue_security_context

This operation is invoked by the association interceptor to continue establishment of the security association. It can be called by either the client or target interceptor on the local security context object.

```
AssociationStatus continue_security_context(
    in      Opaque                  in_token
    out     Opaque                  out_token
);
```

Parameters

in_token	The security token generated by the other one of the client-target pair and sent to this **Security Context** object to be used to continue the dialogue between client and target to establish the security association.
out_token	If required, a further security token to be returned to the other **Security Context** object to continue the dialogue.

Return Value

SecAssocSuccess	The security association has been successfully established.
SecAssocFailure	The attempt to establish a security association has failed.
SecAssocContinue	The context is only partially initialized and further operations are required to complete authentication.

protect_message

The **protect_message** operation of the **Security Context** object provides the means whereby the client message protection interceptor may protect the request message, or the target interceptor may protect the response message for integrity and/or confidentiality according to the Quality of Protection required.

```
void protect_message(
    in      Opaque          message,
    in      QOP             qop,
    out     Opaque          text_buffer,
    out     Opaque          token
);
```

Parameters

message	The message for which protection is required.
qop	Required message protection options.
text_buffer	The protected message, optionally encrypted.
token	The integrity checksum, if any.

Return Value

None.

reclaim_message

The **reclaim_message** operation on the **SecurityContext** object provides the means whereby a protected message may be checked for integrity and the message optionally decrypted if needed.

```
boolean reclaim_message(
    in      Opaque                  text_buffer,
    in      Opaque                  token,
    out     QOP                     qop,
    out     Opaque                  message
);
```

Parameters

text_buffer	The message for which the check is required and optionally the message to be decrypted.
token	The integrity checksum, if any. Will typically be zero length if QOP indicates that confidentiality was applied.
qop	The quality of protection that was applied to the protected message.
message	The unprotected message, decrypted if required.

Return Value

If the **reclaim_message** operation returns a value of FALSE, then the message has failed its integrity check. If it returns a value of TRUE, the message has passed its integrity check.

is_valid

The **is_valid** operation of the **Security Context** object allows a caller to determine whether the context is currently valid.

```
boolean is_valid(
    out     UtcT                    expiry_time
);
```

Parameters

expiry_time	The time at which this context is no longer valid.

Return Value

FALSE	The context is no longer valid.
TRUE	The context is still valid.

refresh

May extend the useful lifetime of the **SecurityContext**. The precise behavior is implementation-specific. The **refresh** operation may be called on both valid and expired contexts.

boolean refresh ();

Parameters

None.

Return Value

If the **refresh** operation returns a value of FALSE, then the context could not be refreshed. In this case, the caller should acquire a new context using the **Vault::init_security_context** interface. If TRUE, the context was successfully refreshed.

15.7.2.5 The Credentials Object

The **Credentials** object with the **SecurityLevel2::Credentials** interface, as defined in Section 15.5.4, "The Credentials Object," on page 15-94, is used to pass **Credentials** information between the underlying security mechanisms and the ORB Security Services.

15.7.2.6 The Access Decision Object

The **Access Decision** object is responsible for determining whether the specified credentials allow this operation to be performed on this target object. It uses access control attributes for the target object to determine whether the principal's privileges, obtained from the **Security Context** are sufficient to meet the access criteria for the requested operation. **Access Decision** objects have the **SecurityLevel2::AccessDecision** Interface as described in "The Access Decision Object" on page 15-118.

15.7.2.7 Audit Objects

There are two types of audit objects:

1. The **Audit Decision** object, which has the **SecurityLevel2::AuditDecision** interface, is used to find out whether an action needs to be audited. Similar audit decision objects are used for all audit policies.
2. The **Audit Channel** object, which has the **SecurityLevel2::AuditChannel** interface, is used by many of the implementation components (such as interceptors and security objects) and also used by applications to write audit records.

The interfaces are described in Section 15.5.7, "Security Audit," on page 15-113.

15.7.2.8 Principal Authentication

The **Principal Authenticator** object with the **SecurityLevel2::Principal Authenticator** interface, defined in Section 15.5.3, "Authentication of Principals," on page 15-90, provides the facility for authenticating a principal. It may also be used by implementation security objects, specifically the **Vault**.

15.7.2.9 Non-repudiation

The Non-repudiation services are accessible through the **NRservice::NRCredentials** interface. Its functionality and operations are defined in Section 15.5.11, "Non-repudiation," on page 15-121.

15.7.3 Replaceable Security Services

It is possible to replace some security services independently of others.

15.7.3.1 Replacing Authentication and Security Association Services

Replacement of the authentication, security context management, and message protection services underlying a secure ORB implementation can be accomplished by replacing the **Principal Authenticator**, **Vault**, **Credentials**, and **Security Context** objects with implementations using the new underlying technology.

Note that if the **Vault** uses GSS-API to link to external security services, it may be substantially security-technology-independent, and so may require no changes or minor changes in order to accommodate a new underlying authentication technology (though it may also have to use technology-independent interfaces for principal authentication in some circumstances, as this is not always hidden under GSS-API).

The Vault is replaced by changing the version in the environment.

15.7.3.2 Replacing Access Control Policies

Access control policies can be changed by replacing the **Access Policy** and **Access Decision** objects, which define and enforce access control policies (for example, substituting another **Access Policy** object for **DomainAccessPolicy**).

Applications may also change their access control policies. If the application access policy object(s) is similar to the invocation access policy object(s), then they can be replaced in a similar way.

15.7.3.3 Replacing Audit Services

Audit policies may be replaced, for example, to support certain types of invocation audit policy not supported by the standard audit policy objects. In this case, the policy objects are replaced in a way that is similar to how the access policy objects are.

Also, **Audit Channel** objects may be replaced to change how audit records are routed to a collection point or filtered.

The **Audit Channel** object used for object system auditing is replaced by replacing the **Audit Channel** object in the environment. Other **Audit Channel** objects may be replaced by associating a different channel object with the appropriate audit policy.

Application auditing objects can be replaced by the application.

15.7.3.4 Replacing Non-repudiation Services

The Non-repudiation Service is a stand-alone replaceable security service associated with **NRCredentials** and **NRPolicy** objects. Different NR services may use different mechanisms and support different policies. For example, it may be that a service using symmetric encipherment techniques may be replaced by a service using asymmetric encipherment techniques.

The same credentials and authentication method may be used for non-repudiation and for other secure invocations, so when replacing either of these, the effect on the other should be considered.

15.7.3.5 Other Replaceability

No other replaceability points are defined as part of this specification. However, individual implementations may permit replacement of other security services or technologies.

15.7.3.6 Linking to External Security Services

The security service interfaces specified in this section may encapsulate calls to external security services via APIs.

The external services used may include:

- Authentication Services, to authenticate principals.
- Privilege (Attribute) Services, for selecting and certifying privilege attributes for authenticated principals (if access control can be based on privileges as well as on individual identity).
- Security Association Services, for establishing secure associations between applications. These services may themselves use other security services such as Key Distribution Services (if secret keys are used), a Certification Authority for certifying public keys, and Interdomain Services for handling communications between security policy domains.
- Audit (and Event) Services.
- Cryptographic Support Facilities, to perform cryptographic operations (perhaps in an algorithm-independent way).

This specification does not mandate which interfaces are used to access external security services, but notes the following possibilities:

- The GSS-API is used for security associations and for the majority of Credentials and Security Context operations, as this allows easy security service replacement. With this in mind, several interfaces in this specification have been designed to allow easy mapping to GSS-API functions, and the Credentials and Security Context objects are consistent with GSS-API credentials and contexts.
- IDUP GSS-API may be used for independent data unit protection and evidence generation and verification.
- Cryptographic operations performed by a Cryptographic Support Facility (CSF) to ease replacement of cryptographic algorithms. No specific interface is recommended for this yet, as such interfaces are being actively discussed in X/Open and other international bodies, and standards are not yet stable.

15.8 Security Interoperability Protocols

15.8.1 Introduction

This section specifies a model for secure interoperability between ORBs which conform to the CORBA 2 interoperability specification and employ a common security technology.

The interoperability model also describes other interoperability cases, such as the effect on interoperability of crossing security policy domains. However, detailed definitions of these are not given in this specification.

It then defines the extensions required to the interoperability protocol for security. This includes:

- specification of tags in the CORBA 2 Interoperable Object Reference (IOR) so this can carry information about the security policy for the target object and the security technology which can be used to communicate securely with it.
- a security interoperability protocol to support the establishment of a security association between client and target object and the protection of CORBA 2 General Inter-ORB Protocol (GIOP) messages between them for integrity and/or confidentiality. This is independent of the security technology used to provide this protection.
- security when using the DCE-CIOP protocol.

As the security information needed by a security mechanism is generally independent of which ORB interoperability protocol is used, other Environment-Specific Protocols (ESIOPs) may support security in a similar way to that described for GIOP. However, the specification only addresses DCE-CIOP, which supports only DCE security.

The security protocol specified does not define details of the contents of the security tokens exchanged to establish a security association, the integrity seals for message integrity, or the details of encryption used for confidentiality of messages, as these depend on the particular security mechanism used. This specification does not specify mechanisms.

15.8.2 Interoperability Model

This section describes secure interoperability when:

- the ORBs share a common interoperability protocol,
- consistent security policies are in force at the client and target objects, and
- the same security mechanism is used.

All other options build from this.

The model for secure interoperability is shown in the following diagram.

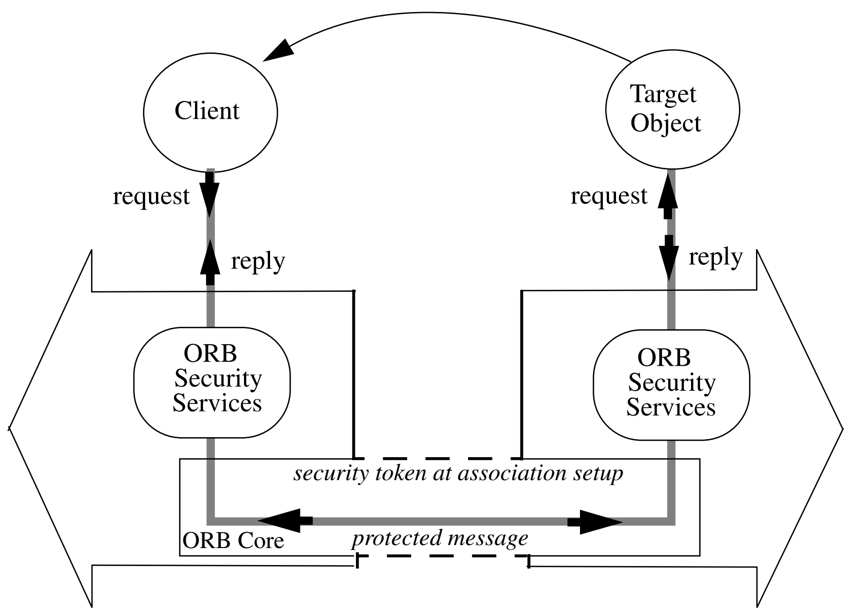


Figure 15-55 Model for Secure Interoperability

When the target object registers its object reference, this contains extra security information to assist clients in communicating securely with it.

The protocol between client and target object on object invocations is as follows:

- If there is not already a security association between the client and target, one is established by transmitting security token(s) between them (transparently to the application).

- Requests and responses between client and target are protected in transit between them. Protection includes not only ensuring individual messages are inviolate and private, but that message streams are as well.

15.8.2.1 Security Information in the Object Reference

When an object is created in a secure object system, the security attributes associated with it depend on the security policies for its domain and object type and the security technology available. A client needs to know some of this information to communicate securely with this object in a way the object will accept. The object reference transferred between two interoperating systems includes the following information:

- A security name or names for the target so the client can authenticate its identity.
- Any security policy attributes of the target relevant to a client wishing to invoke it. This covers policies such as the required quality of protection for messages and whether the target requires authentication of the clients identity and supports authentication of its identity.
- Identification of the security technology used for secure communication between objects this target supports and any associated attributes. This allows the client to use the right security mechanism and cryptographic algorithms to communicate with the target.

15.8.2.2 Establishing a Security Association

The contents of the security tokens exchanged depend on the security mechanism used.

A particular security mechanism may itself have options on how many security tokens are used. The minimum is an *initial context* token (a term used in GSS-API), sent from the client to the target object to establish the security association. This typically contains:

- an identification of the security mechanism used,
- security information used by this mechanism to establish the required trust between client and target and to set up the security context necessary for protecting messages later,
- the principal’s credentials, and
- information for protecting this security data in transit.

In addition to this token, subsequent security tokens may be needed if:

- mutual authentication of client and target object is required, or
- some negotiation of security options for this mechanism is required (for example, the choice of cryptographic algorithms).

15.8.2.3 Protecting Messages

The invocation may be protected for integrity and/or confidentiality. In either case, the messages forming the request and reply are first wrapped in a sequencing layer envelope and then cryptographically protected by the ORB security services. For integrity, extra information (e.g., an integrity seal) is added to the message so the target ORB security services can check that the message has not been changed.

For confidentiality, the message itself is encrypted so it cannot be intercepted and read in transit.

Details of how messages are protected are again mechanism-dependent. Note, however, that messages cannot be changed once they have been protected, as they cannot be understood once confidentiality protected and the integrity check will fail if they are altered in any way.

In SECIOP message stream protection is provided by encapsulating all SECIOP data payloads (e.g., IIOP messages or message fragments) in a sequencing protocol frame. The sequencing protocol ensures that data payloads are not duplicated (replayed), dropped (deleted), or received out-of-sequence (reordered). The sequencing protocol frame is protected by the ORB security services to ensure the state it contains is not modified by an intruder.

15.8.2.4 Security Mechanisms for Secure Object Invocations

The interoperability model above can be supported using different security mechanisms.

This specification does not define a standard security mechanism to be supported by all secure ORBs. It therefore does not specify a particular set of security token formats and message protection details for a particular security mechanism.

15.8.2.5 Security Mechanism Types

There are two major types of security mechanisms used in existing systems for security associations. They are those using:

- Symmetric (secret) key technology where a shared key is used by both sides, and a trusted third party (a Key Distribution Service) is used by the client to obtain a key to talk to the target.
- Asymmetric (public) key technology where the keys used by the two sides are different, though linked. In this case, long term, public keys are normally freely available in certificates which have been certified by a Certification Authority.

Several existing systems use symmetric key technology for key distribution when establishing security associations. These are usually based on MIT's Kerberos product. Such systems normally include no public key technology.

Other security mechanisms use public key technology for authentication and key distribution as this has advantages for scalability and inter-enterprise working. The number of public key-based systems are growing and the use of public key technology is standard for non-repudiation, which is an optional component in this specification, and increasingly needed in commercial systems so any OMG security specification must not preclude its use. Also, the use of smart cards with public key technology is increasing. However, non-repudiation is not a service required for secure interoperability.

Interoperating with Multiple Security Mechanisms

The current specification allows a client to identify the security mechanism(s) supported by the target. Where a client or target supports more than one mechanism, and there is at least one mechanism in common between client and target, the client can choose one which they both support.

Some security mechanisms may support a number of options, for example:

- a choice of cryptographic algorithms for protecting messages,
- a choice of using public or secret key technology for key distribution.

The appropriate options can be chosen by the client in the same way as choosing the basic mechanism, via the client security policy and information in the target's object reference. However, some mechanisms will be able to negotiate options using extra exchanges at association establishment which are specific to the particular mechanisms.

Interoperating between Underlying Security Services

Security mechanisms for secure object invocations use underlying security services for authentication, privilege acquisition, key distribution, certificate management, and audit. Under some circumstances, these need to inter-operate. For example, key distribution services may need to communicate with each other, and audit services may need to transmit audit records between systems.

Interoperability of such underlying security services is considered out of scope of this specification, as they are mechanism-dependent.

15.8.2.6 *Interoperating between Security Policy Domains*

The sections above consider interoperability within a security policy domain where consistent security policies apply to access control, audit and other aspects of the system. These rely on information about the principal, including its identity and privilege attributes, being trusted and having a consistent meaning throughout the policy domain.

Where a large distributed system is split into a number of security policy domains, interoperation between security policy domains is needed. This requires the establishment of trust between these domains. For example, an ORB security association service at a target system will need to identify the source of the principal's credentials so it can decide how much to trust them.

Once the identity of the client domain has been established, interdomain security policies need to be enforced. For example, access control policies are mainly based on the principal's certified identity and privilege attributes. The policy for this could be:

1. The target domain trusts the client domain to identify principals correctly, but does not trust their privilege attributes, so treats all principals from other domains as guest users.
2. The administrators of the two domains have agreed some privilege attributes in common, and trust each other to give these only to suitably authorized users. In this case, the target system will give principals from the client domain with these privileges the same rights as principals from the target domain.
3. The administrators of the two domains agree what particular privilege attributes in the client domain are equivalent to particular privilege attributes in the target domain, and so grant corresponding access rights.

For the first two of these, the target domain security policy could enforce restrictions about which privilege attributes may be used there. This would not necessarily affect the interoperability protocols - the **get_attributes** operation will simply not return all of the privileges. But even in this case, some security mechanisms will choose to modify the principal's credentials to exclude unwanted attributes.

In the third case, the privilege attributes need to be translated to the ones used in the target domain. If this translation is to be done only once, an interdomain service is likely to be used which both translates the credentials and reprotects them so they can be delegated between nodes in the target domain.

Such an interdomain service may be invoked by the ORB Security Services, but may be invoked by a separate interoperability bridge between the ORB domains. If invoked by an ORB service, it extends the implementation of the **Vault** object described previously and this will probably call on a mechanism-specific Interdomain Service.

15.8.2.7 Secure Interoperability Bridges

Secure Interoperability Bridges between ORB domains are relevant to this architecture, as in the future, they may be specified as part of some secure CORBA-compliant systems. However, this section does not describe how to build such bridges.

Secure interoperability bridges may be needed for:

- ORB-mediated bridges, where data marshalling is done outside the ORB and associated ORB services.
- Translating between security mechanisms (technology domains).
- Mapping between security policy domains.

In all these cases, both the system and application data being passed will need to be altered, affecting its protected status. This needs to be re-established using security services trusted by both client and target domains.

15.8.3 Protocol Enhancements

The following sections detail the enhancements required to the CORBA 2 interoperability specification for security.

Section 15.8.4, "CORBA Interoperable Object Reference with Security," on page 15-178 defines the enhancements needed to the Interoperable Object Reference (IOR).

Section 15.9, "Secure Inter-ORB Protocol (SECIOP)," on page 15-204 defines the enhancements needed to secure GIOP messages and Section 15.15, "DCE-CIOP with Security," on page 15-273 defines the DCE-CIOP with security.

15.8.4 CORBA Interoperable Object Reference with Security

The CORBA 2 Interoperable Object Reference (IOR) comprises a sequence of 'tagged profiles'. A profile identifies the characteristics of the object necessary for a client to invoke an operation on it correctly, including naming/addressing information. The tag is a standard, OMG-allocated identifier for the profile which allows the client to interpret the profile data, but although the tag is OMG-allocated, the profile itself may not be OMG-specified.

A multi-component profile is a profile that itself consists of tagged components. This specification defines TAGS for use in such multi-component profiles as follows:

The following TAGs are defined:

- **IIOP components**, which can be used in a multi-component profile (see Appendix Section A.16, "Secure Inter-ORB Protocol (SECIOP)," on page 15-301).
- **Security components** that identify security mechanism types, one for each mechanism supported. Each security mechanism component can also include mechanism-specific data.
- Aspects of the target object policy that cover the dependencies between and overall use of components (for example, the quality of protection required) may be specified in separate **policy components**. This avoids establishing unnecessary dependencies between other (technology) components.

Use of tagged components within the multi-component profile to carry IIOP, security and other data may cause performance degradations in certain situations. For example, if an IOR carries many tagged components that are unrecognized by a client implementation, it must process these when they appear before those that it does recognize. Some, such as the components describing IIOP, have a high probability of being recognized and used by many clients. Consequently, implementations with an objective to optimize IOR processing will place such components at the beginning of the tagged component sequence.

15.8.4.1 Security Components of the IOR

The following new tags are used to define the security information required by the client to establish a security association with the target. Note that a tag may occur more than once, denoting that the target allows the client some choice. All tag component data must be encapsulated using CDR encoding

TAG_x_SEC_MECH

This is the prototype TAG definition for OMG-registered security association mechanisms. The mechanism is identified by the TAG value. The component data for TAGs of this kind is defined by the person who registers the TAG. The confidentiality and integrity algorithms to be used with the mechanism may either be encoded into the TAG value or in mechanism-specific data (see Appendix Section G.2, "Guidelines for Mechanism TAG Definition in IORs," on page 15-368).

If this definition includes:

sequence <TaggedComponent> components;

The components field can contain any of the other component TAGs, whose values can be specific to the mechanism.

If the mechanism is selected for use, the components in this field are used in preference to any recorded at the multi-component level.

Multiple **TAG_x_SEC_MECH** components may be present to enumerate the security mechanisms available at the target.

TAG_GENERIC_SEC_MECH

This TAG enables mechanisms not registered with the OMG, but common to both client and target to be used with the standard interoperability protocol. Its definition is:

```
struct GenericMechanismInfo {
    sequence <octet> security_mechanism_type;
    sequence <octet> mech_specific_data;
    sequence <TaggedComponent> components;
};
```

The first part of this TAG is the **security_mechanism_type** which identifies the type of underlying security mechanism supported by the target including confidentiality and integrity algorithm definition. It is an ASN.1 Object Identifier (OID) as described for use with the GSS-API in IETF RFC 1508.

The **mech_specific_data** field allows mechanism-specific information to be passed by the target to the client.

The components field can contain any of the other component TAGs, whose values can be specific to the mechanism.

If the mechanism is selected for use, the components in this field are used in preference to any recorded at the multi-component level.

Multiple **TAG_GENERIC_SEC_MECH** components may be present to enumerate the security mechanisms available at the target.

TAG_ASSOCIATION_OPTIONS

This TAG is used to define the association properties supported and required by the target. Its definition is:

```
struct TargetAssociationOptions{
    AssociationOptions    target_supports;
    AssociationOptions    target_requires;
};
```

target_supports - gives the functionality supported by the target.

target_requires - defines the minimum that the client must use when invoking the target, although it may use additional functionality supported by the target.

The following table gives the definition of the options.

Table 15-10 Definition of Association Options

Association Options	target_supports	target_requires
NoProtection	the target supports unprotected messages	the target's minimal protection requirement is unprotected invocations
Integrity	the target supports integrity- protected messages	the target requires messages to be integrity-protected
Confidentiality	the target supports confidentiality-protected invocation	the target requires invocations to be protected for confidentiality
DetectReplay	the target can detect replay of requests (and request fragments)	the target requires security associations to detect message replay
DetectMisordering	target can detect sequence errors of requests and request fragments	the target requires security associations to detect message mis-sequencing
EstablishTrustInTarget	the target is prepared to authenticate its identity to the client	(this option is not defined)
EstablishTrustInClient	the target is capable of authenticating the client	the target requires establishment of trust in the client's identity

TAG_SEC_NAME

The target security name component contains the security name used to identify and authenticate the target. It is an octet sequence, the content and syntax of which is defined by the authentication service in use at the target. The security name is often the name of the environment domain rather than the particular target object.

The **TAG_SEC_NAME** component is not needed if the target does not need to be authenticated.

15.8.4.2 *IOR Example*

Table 15-11 IOR Example

tag	value	mech specific tag	value
tag_sec_name	"Manchester branch"		
tag_association_options	supports and requires integrity and to establish trust in the client's privileges		
tag_generic_sec_mech	mech 1 oid		
		tag_sec_name	"MBn1"
		tag_association_options	supports and requires integrity, replay detection, misordering detection, and to establish trust in the client's security attributes
tag_generic_sec_mech	mech 2 oid		
		tag_association_options	target requires and supports confidentiality and to establish trust in the client's security attributes

In this example, if mechanism "mech 1" is used, the target security name is "MBn1" while the association must use integrity replay and misordering options. If mechanism "mech 2" is used, no mechanism-specific security name has been specified and so "Manchester branch" is used as the security name. The association options are EstablishTrustInClient and Integrity.

15.8.4.3 *Operational Semantics*

This section describes how an ORB and associated ORB services should use the IOR security components to provide security for invocations and how the target object information should be provided.

Client Side

During a request invocation, the non-security tagged components in the IOR multi-component profile indicate whether the target supports IIOP and/or some other environment-specific protocol such as DCE-CIOP. Security mechanism tag components specify the security mechanisms (and associated integrity and confidentiality algorithms) this target can use. The ORB selects a combination of interoperability protocol and security mechanism which it can support.

If there is a common interoperability protocol, but no common security mechanism, then a secure request on this IOR cannot be assured.

If the same security mechanism is supported at the client and the target, but the **TAG_ASSOCIATION_OPTIONS** component specifies no protection is needed or no **SEC_MECH** is specified, then unprotected requests are supported by the target, and the request can be made without using security services. If the target requires protected requests, then the ORB must choose an alternative transport and/or security mechanism.

The IOR tags and the client's policies and preferences are used together to choose the security for this client's conversation with the target.

The specific security service used may not understand the CORBA security values, and so may require them to be mapped into values it can understand.

Determining Association Options

The **Association Options** in Table 15-10 on page 15-180, lists possible association options such as **NoProtection**, **Integrity**, **DetectReplay**.

The actual association options used when a client invokes a target object via an IOR depend on:

- The client-side secure invocation policy and environment.
- Client preferences as specified by **set_association_options** on the **Credentials** or **set_policy_overrides** of the object reference invoked with a **QOPPolicy** object as one of the Policies to be overridden.
- The target-side secure invocation policy and environment (as indicated by information in the **TAG_ASSOCIATION_OPTIONS** component).

An association option should be enforced by the security services if the client requires it and the target supports it, or the target requires it and the client supports it.

If the target cannot support the client's requirements, then a CORBA::NO_PERMISSION exception should be raised. If the client cannot meet the requirements of the target, then the invocation may optionally proceed, allowing policy enforcement on the target side.

Target Side

The security information required in the IOR for this target must be supplied from the target (or its environment). This specification does not define exactly when particular information is added, as some of it may only be needed when the object reference is exported from its own environment.

The security information may come from a combination of:

- The object's *own credentials* (see "Security Operations on Current" on page 15-105). This includes for example, the target's security name. It could include mechanism-specific information such as the target's public key if it has one.
- Policy associated with the object. This includes, for example, the QOP.
- The environment. This includes, for example, the mechanism types supported.

The target object does not need to supply this information itself. This is done automatically by the ORB when required. For example, much of the information for the target's own credentials are set up on object creation.

As at the client, the specific security service used may require CORBA security values to be mapped into those it understands.

If when the client invokes the target identified by the IOR an **Invoke Response** message is returned for the request with the status **INVOKE_LOCATION_FORWARD**, then the returned multiple component profile must contain security information as well as the new binding information for the target specified in the original **Invoke Request** message.

Any security information in the returned profile applies to the new binding information and replaces all security information in the original profile. This **INVOKE_LOCATION_FORWARD** behavior can be used to inform the client of updated security information (even if the address information hasn't changed).

15.8.5 Common Secure Interoperability Levels

Three Common Secure Interoperability Levels are defined to help in classifying and positioning the various interoperability facilities that are defined, and also to help in concisely stating the conformance requirements. The three CSI levels are:

CSI Level 0 - supports only identity-based policies without delegation.

CSI Level 1 - supports identity-based policies with or without unrestricted delegation.

CSI Level 2 - supports identity- and privilege-based policies with controlled delegation.

A complete description of the these CSI levels of interoperability can be found in Appendix Section C.7.2, "Common Secure Interoperability Levels," on page 15-326.

15.8.6 Key Distribution Types

Security mechanisms use cryptography in the establishment of a secure association between a client and target and in protecting the data between them. Security mechanisms differ in the type of cryptography they use, particularly for distribution of keys. (Keys are assigned to clients, targets, and trusted authorities). Three types of key distribution are defined in this specification:

- ***Secret keys*** - use secret key technology for distribution of keys for principals.
- ***Public keys*** - use public key technology for distribution of keys for principals, though may use secret key technology for message protection.
- ***Hybrid*** - use secret key technology for key distribution for principals within an administration domain, and public key technology for key distribution for trusted authorities, and hence between domains.

All types of key distribution can be used to support all the facilities in CORBA Security for secure object invocations (though public key is almost universally used for non-repudiation). The choice of mechanism to use depends on a customer's requirements. For example, to fit with other systems and for scalability to inter-enterprise working.

15.8.7 Security Mechanisms Hosted on SECIOP

Choosing the protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. How the mechanisms underlying the following three security protocols are hosted on SECIOP are specified:

15.8.7.1 SPKM Protocol

Supports identity-based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20].

15.8.7.2 GSS Kerberos Protocol

Supports identity-based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0).

The GSS Kerberos protocol is based on [12] which itself is a profile of [13].

15.8.7.3 CSI-ECMA protocol

Supports identity- and privilege-based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0).

For keys assigned to principals, it has two options:

- It can use either secret or public key technology.
- It uses public key technology for keys assigned to trusted authorities.

The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235.

Table 15-12 shows which CSI functionality is supported with which protocols.

Table 15-12 CSI Functionality and Protocols

Protocol / CSI Level	SPKM	GSSKerberos	CSI-ECMA
0	Supported	Supported	Supported
1	Not supported	Supported (Mandatory)	Supported
2	Not supported	Not supported	Supported

15.8.8 Security Mechanisms Hosted Directly on IIOP

The SSL [21] protocol which provides for confidentiality and integrity within the IP sockets paradigm can be used to provide interoperability based on this protocol hosted directly on IIOP. How this is done is specified in Section 15.14, "Integrating SSL with CORBA Security," on page 15-272. It supports identity-based policies without delegation.

15.8.9 Choices of Protocols, Cryptographic Profiles, and Key Technologies

What combination of Security Protocols, Key Technologies, and Cryptographic Profiles are the most desirable has been subject of debate both inside and outside OMG. In this specification, certain choices have been made based on the belief that these choices best meet OMG's current needs given the other constraints.

15.8.9.1 Choice of Protocol and Key Technology

GSS Kerberos is specified as the mandatory protocol for common secure interoperability, as Kerberos is widely available and most vendors can support it. However, it does not provide all facilities required and is secret key only.

Several other protocols are specified as non-mandatory options as follows:

- CSI-ECMA is specified as a protocol to provide support for the full set of CORBA security facilities using public key or secret key technology.
- SPKM is specified as a simpler public key protocol suitable for applications where
  - access and audit policies are static, and
  - at each stage in a chain of object invocations, the policies depend only on the identity of the immediate invoker, not the initiator of the chain.

- SSL is specified for use in the web market.

15.8.9.2 Cryptographic Profiles

Security mechanisms use cryptography in the establishment of a secure association between a client and target and in protecting the data between them. Different cryptographic algorithms are used to support particular security functions depending on the type of mechanism used and also the regulations on use of cryptography. The combination of algorithms used to provide particular security using a particular mechanism is called a cryptographic profile.

Currently, different cryptographic algorithms, and/or different key lengths are required to meet export controls and regulations on use of cryptography in various countries (see “International Deployment” on page 15-187). Although some vendors produce more than one version of secure products for different markets, they are increasingly reluctant to do this. For common secure interoperability, a particular cryptographic profile is needed. Some options are to standardize:

- Integrity only for user data, not confidentiality. If done using MD5, this is likely to be exportable and generally deployable, but doesn’t provide confidentiality when interoperating. This does not provide the functionality which some users will want.
- Integrity and confidentiality using weak keys only. This provides the required functionality, in a way which can generally be exported, but does not provide the strength of protection needed by some customers. Also, products using it may be subject to import controls or other regulations in some countries.
- On strong confidentiality and integrity, which customers want, but will be subject to export controls in most countries and to deployment regulations in some. Leave vendors and customers to sort out the problems.

This chapter makes only the first of these options mandatory; however, implementors of all profiles may choose to support other profiles also.

15.8.9.3 Conformance to External Security Mechanisms

This specification uses protocols defined in other standards documents. It refers to particular versions of these standards, which is needed for interoperability. If the versions of these external documents change in the future, there may be a need to update this specification so that it is in line with the most accepted external version of these standards.

15.8.10 Common Secure Interoperability Requirements

This section describes the requirements that Common Secure Interoperability is expected to meet.

The Common Secure Interoperability specification is required to provide for standard security mechanisms, simple delegation, and international deployment. This section discusses the key requirements for common secure interoperability that have driven the design of this specification and how this specification responds to these requirements.

15.8.10.1 CORBA Standard Security Mechanisms

Standard CORBA security mechanisms are required so that ORBs can interoperate securely.

Four popular security mechanisms to meet different circumstances, as described above, can be used to host CORBAsecurity in a standard way. One of the four described in this chapter is mandatory and all conformant ORBs must support it. Interoperability between conformant ORBs is always possible using this; however, the facilities supported when using it are limited.

Interoperability also requires common use of cryptographic algorithms. A number of cryptographic profiles are specified to meet the needs of different markets and countries. One is mandatory and interoperability between conformant ORBs is always possible using this; however, it provides data integrity but not confidentiality.

Where multiple mechanisms and cryptographic profiles are supported by both ORBs, the client and target object must agree which to use. In this specification, this is done by the client looking at the security mechanism tag in the target object reference and choosing an appropriate mechanism and profile which both support. (In the future, negotiation of mechanisms may be supported.)

15.8.10.2 International Deployment

International deployment requires that the security mechanisms and algorithms chosen can be used worldwide in countries which are subject to different national regulatory controls on the use of cryptography. It also requires that they can be used across international boundaries. International deployment may also be affected by export control regulations and other issues.

Requirements distilled from the key regulations affecting international deployment include:

- Keeping the amount of information which must be encrypted for confidentiality to a minimum. In general, encryption of keys is acceptable, but encryption of other data may not be. For this reason, encryption of security attributes is undesirable. At CSI level 2, where more attributes are generally needed, the part of the security tokens concerned with key distribution is separated from the part used to carry privileges (e.g., in CSI-ECMA); therefore, the latter part does not have to be encrypted.
- Being able to use identities for auditing which are anonymous, except to the auditor. For this reason, identities used for access control and audit may need to be different. A separate **AuditId** can be transmitted at level 2.
- Allowing use of different cryptographic algorithms, with different lengths of keys for specified functions to meet export and use regulations in different countries. The specification defines cryptographic profiles which allow for different cases. The mandatory one provides data integrity only, as this is generally easier to deploy internationally.

There may be further requirements on secure ORB products to ensure that they are exportable. For example, they must not allow easy/uncontrolled replacement of cryptographic algorithms. This affects the construction of the system, but not this interoperability standard, so this is not considered further in this specification.

Other restrictions on the use of algorithms and security mechanisms are highlighted in "Identifying Encumbered Technology" on page 15-190. For example, the DES algorithm is subject to export controls, while RSA requires licensing in some countries. The MIT version of the Kerberos technology, widely used in the USA, is also subject to export controls.

15.8.10.3 *Consistency*

It should be possible to provide consistent security across the distributed object system and with associated legacy and other non-object systems. This includes:

- Support of consistent policies for which principals should be able to access the sort of information, within a security domain, that includes heterogeneous systems.

  For this specification, it requires the ability to transmit consistent privilege and other attributes between ORBs to support these policies. Level 0 and 1conformant ORBs can transmit identities, level 2 conformant ORBs can transmit a range of privilege attributes. These can be the ones used in existing systems, though system-specific ones will not be usable in other systems.

- Fit with existing logons (so extra logons are not needed) and with existing user databases (to reduce the user administration burden).

  Log on needs to result in credentials which include the information required to support the specified security mechanisms. Note that single logon with secure messaging, web, etc. generally requires use of public key-based mechanisms. Also, if non-repudiation is supported, they will also need to include the security information required to support the non-repudiation mechanism (normally, a public key mechanism).

  Also, interoperating with non-object systems may require, for example, a CORBA object implementation which calls a non-CORBA application to be able to delegate incoming credentials (assuming compatible security mechanisms.)

- Fit with all non-object systems is clearly not possible if such a system uses security mechanisms which are incompatible with the one used in the object system. Such systems may be able to use CORBA Security, but will not be able to interoperate using the common secure interoperability standard.

This specification includes an interoperability level which supports privileges and a public key (as well as a secret key) mechanism to support these requirements.

15.8.10.4 *Scalability*

It should be possible to provide security for a range of systems from small, local systems to large intra- and inter-enterprise systems. For larger systems, it should be possible to:

- Base access controls on the privilege attributes of users such as roles or groups (rather than individual identities) to reduce administrative costs. This specification includes the transmission of such privilege attributes in CSI level 2.
- Have a number of security domains which enforce different security policy details, but support interworking between them subject to policy. (This specification includes the architecture for such inter-domain working, though this specification does not define an interface for this.) Use of public key technology helps large-scale, particularly inter-enterprise interoperability.
- Manage the distribution of cryptographic keys across large networks securely and without undue administrative overhead.

15.8.10.5 Flexibility of Security Policy

The security policies required vary from enterprise to enterprise, so choices should be allowed, though standard policies should be supported for common secure interoperability.

Access Policies

At CSI levels 0 and 1, the **AccessId** is the only privilege attribute supported. The standard **DomainAccessPolicy** defined in Section 15.6.4, "Access Policies," on page 15-133 (or other access policies) can be used with only this privilege.

At CSI level 2, conformant ORBs are able to transmit further privilege attributes (such as role and group), so the **DomainAccessPolicy** (and other access policies) can be used with these privileges also.

CSI level 2 is designed to allow transmission of further privileges, including user-defined privileges and security clearances as needed for multi-level secure systems. If received by a conformant ORB, they will be available for access control at the target. However, conformant ORBs need not transmit them, so use of such privileges is subject to the agreement between the systems.

The mechanisms defined here also allow a wider range of privileges, etc., to be supported and other access policies to be used. However, interoperability with all other conformant ORBs is not guaranteed in this case.

Audit Policies

All CSI levels provide an **AuditId** which can be used in audit policies. CSI level 2 can transmit an **AuditId** which is anonymous to all but audit administrators.

15.8.10.6 Application Portability

Application portability is an important OMG requirement. The many applications which are unaware of security will continue to be portable.

Applications which enforce their own security policies should still be portable across ORBs supporting common secure interoperability if the access and audit policies they use rely only on security attributes which are mandatory in the chosen CSI level.

Applications should be unaware of the security mechanism used to enforce the security, unless they specifically ask what it is (e.g., using **get_service_information**, see Section 15.5.2, "Finding Security Features," on page 15-90).

15.8.10.7 Security Services Portability/Replaceability

The CORBA Security specification includes replaceability conformance options.

The objects supporting the security mechanism (**PrincipalAuthenticator**, **Vault**, and **Security Context**) can be replaced to support the mechanisms in this specification. However, if logon outside the object system is supported, this will need to provide credentials including the security information needed by the CSI mechanism(s) used.

If the invocation access policy is replaced, this can utilize privileges transmitted using CSI protocols. However, if an ORB wishes to control access on invocations using local (e.g., operating system) attributes, then mapping of attributes prior to calling the **Access Decision** object is needed.

15.8.10.8 Performance

Security should not impose an unacceptable performance overhead, particularly for normal commercial levels of security, although a greater performance overhead may occur as higher levels of security are implemented.

Details of the performance overhead depend on the mechanism used and its implementation; however, in this specification:

- Sufficient information can be carried in the IOR so that the client knows what security the target supports and does not have to negotiate protocols and options with it.
- The mechanisms used allow the **initial_context_token** to be transmitted with first message, if mutual authentication is not required.

15.8.10.9 Identifying Encumbered Technology

This specification includes technology which is encumbered to some extent.

- The Kerberos V5 technology is licensable from the Massachusetts Institute of Technology without cost and is widely deployed within the USA. However, it is subject to export control from the USA; therefore, [12] is the definition of the protocol used here, as this can be implemented independently of the MIT Kerberos code.
- SPKM implementations are available, though not free. As for other mechanisms, the (draft) standard is the basis of this specification.

- SESAME implementation is available, but is not free for commercial use, and has restrictions on cryptography for export reasons (the public version does not include commercial cryptographic profiles - it has the secret key algorithm replaced by XOR for export control reasons).
- There are two patents associated with the CSI-ECMA protocol. These are usable free of charge for implementations conformant with this specification under fair conditions (formal definition of these are available from Bull and ICL).
- The DES algorithm is widely deployed internationally, but is subject to export controls. Export with key lengths which provide strong confidentiality is not generally permitted.
- Increasingly, the RSA algorithm is widely deployed internationally; however, it is subject to licensing in the USA. It is also subject to export controls, though where it can be shown that it is not used for confidentiality, products using it are more likely to be exportable.
- Any other cryptographic algorithms used are generally subject to export controls, as is any interface which makes it easy to replace algorithms.

15.8.11 Relation to CORBA Security Facilities and Interfaces

This section describes how the security facilities and interfaces defined in Sections 5.5 through 5.7 map to various elements of security protocol mechanisms. It is aimed at:

- Object implementors developing applications using a secure object system who need to know what security is available.
- Implementors of security policies who may be constrained by the security attributes available when interoperating according to this standard.
- ORB implementors supporting replaceable security policies.

15.8.11.1 Functionality

The security information that is transmitted between ORBs, and which security facilities and policies are supported in an interoperable environment, is described in these sections. Three levels of secure interoperability are defined specifying the particular security attributes that conformant ORBs must support.

Note that the interoperability defined here is for interoperability of requests/responses between ORBs. It does not include interoperability of the evidence tokens used for non-repudiation.

15.8.11.2 Replaceability

In replaceability, options which allow ORB implementors to support a wide range of security policies and mechanisms is defined. For example, the standard **DomainAccessPolices** can be replaced by other policies where ORBs support the appropriate replaceability option. This specification still allows this replaceability, though the policy being added may be restricted by the security information guaranteed to be available.

This specification allows replaceability of security mechanisms by replacement of the **Vault** and **Security Context** objects. It specifies mechanisms and protocols which can be implemented via a GSS-API interface. This adds the potential for having a single implementation of the **Vault** and **Security Context** objects, which by using GSS-API, would be able to use different security mechanisms.

15.8.11.3 Levels of Interoperability

This specification includes three interoperability levels, as described more completely in Appendix C, Section C.7.2, "Common Secure Interoperability Levels," on page 15-326. This section gives information about these levels and an example showing the difference how they handle a particular problem.

Common Secure Interoperability Level 0

CSI level 0 supports identity-based policies without delegation. It requires ORBs to support the following:

- Authentication of principals using security functions under one ORB, and then use of the resultant credentials when making a secure invocation to an object under a different ORB.
- Secure associations to establish trust between client, target, and protect messages.
- As part of the secure association, the security name of the client is passed to the target and used to set both **AccessId** and **AuditId** so that identity-based access and audit policies can be supported.

  The identity is always that of the immediate invoker of an object in a chain of object invocations, this is only the same as the initiator of the chain at the point of entry to the chain.

Common Secure Interoperability Level 1

CSI level 1 supports identity-based policies with unrestricted delegation. It requires ORBs to support the mandatory part of the CORBA Security when two conformant ORBs interoperate (using the same security mechanism). It provides the CSI level 0 facilities plus security information (in particular, the security name) of a principal in the call chain can be delegated to objects (subject to security policy).

Once this security information has been delegated, the intermediate object has the choice of acting under its own identity or delegating the initiating principal's identity when invoking another object. When delegating another principal's identity, the delegated identity (rather than the immediate invoker's identity) is used to set both the **AccessId** and **AuditId** at the target.

Common Secure Interoperability Level 2

CSI level 2 supports identity- and privilege-based policies with controlled delegation. ORBs supporting this level must support interoperability of all facilities in Sections 5.5 through 5.7 concerned with object invocation. CSI level 2 provides the CSI level 0 and level 1 facilities plus:

- The security information of the immediate invoker or the delegated information of the initiating principal can include more security attributes, as follows:
  - an extensible range of privilege attributes (e.g., roles, groups, enterprise-defined attributes) to support a wider range of policies. Generally, these attributes include an **AccessId** which is independent of the security name (and the mechanism type used) and is used to set the **AccessId** at the target. Interoperability using particular types of privileges depends on these privileges being common to both ORBs. This CSI specification defines which privileges a CSI level 2 conformant ORB must support (see Appendix Section C.7.2, "Common Secure Interoperability Levels," on page 15-326).
  - a separate **AuditId** can be transmitted. This may be anonymous (except to the audit administrator). It will always represent the actual principal using the system, even when the **AccessId** represents someone who has allowed another user to access the system on his behalf.
- The delegation of a principal's attributes can be controlled (for example, usable at only identified (groups of) targets). Intermediates receiving delegated security attributes of a principal will not always be able to delegate them.
- Composite delegation is allowed for, but support for this is not mandatory.

Example

This section looks at an example of a secure object system which highlights the difference between the delegation facilities of the three CSI levels. In this example, Bob wants to close his bank account and is prepared to give Dan power of attorney to do this.

- At CSI level 0, no delegation is possible; therefore, Bob has to go to the bank and close the account himself.
- At CSI level 1, Bob gives Dan unlimited power of attorney to act for him (as delegation is unrestricted). Dan can close Bob's bank account. As the power of attorney is unlimited, Dan can also read Bob's medical records and pass on the power of attorney to Mark - who can also close Bob's bank account, read Bob's medical records, etc.
- At CSI level 2, Bob gives Dan the power of attorney to close his bank account; therefore, Dan can close the account. But this does not include the right to read Bob's medical records (as only limited privileges were given to Dan) and does not include the right to give the power of attorney to Mark (as delegation was restricted to Dan).

15.8.12 Security Functionality

This section reviews the security functionality in Section 15.5 through 15.7 and specifies which functionality is supported interoperably at which CSI level. Some security functionality is supported at all CSI levels, some only at CSI level 1 or 2.

15.8.12.1 Authentication

The CSI mechanisms do not specify authentication of principals, but use the result of such authentication. Principal authentication must result in credentials which contain the security information needed by the security mechanisms supported by this conformant ORB.

CSI mechanisms require authenticated principals (see Section 15.5.3, "Authentication of Principals," on page 15-90).

15.8.12.2 Access Control

Access controls depend upon the privileges of the principal.

At CSI levels 0 and 1, only the principal's identity is available at the target; therefore, Access Policies using this level must either:

- use only the principal's identity for access control, or
- retrieve other attributes for that principal prior to taking the access decision (the "pull" model).

The standard **DomainAccessPolicy** assumes all privileges required have been "pushed" from the client; therefore, they will be restricted to using identity only. Access policies using the pull model will not be portable, if the source of such attributes is system dependent.

At CSI level 2, the **AccessPolicies** can use any of the privileges supported by both ORBs. All CSI level 2 conformant ORBs support **AccessId**, **GroupId**, and **Role**. They may also transmit user-defined privileges, where the user enterprise concerned has a CORBA attribute family definer, and defines its own families of attributes. However, some attribute types defined outside the object system may not be understood at all targets; therefore, portability of these may not be possible to all environments.

15.8.12.3 Audit

Auditing is defined in Section 15.3.5, "Auditing," on page 15-28, and is possible at all CSI levels. A separate **AuditId** (which may be anonymous) can be transmitted at CSI level 2.

15.8.12.4 Secure Invocation

Conformant implementations (all CSI levels) must support all the association options defined in Table 15-10 on page 15-180.

Channel bindings, as defined in GSS-API and all protocols defined here, are not part of the mandatory specification.

Conformant implementations at level 2 allow use of algorithms with different strengths for integrity and confidentiality.

15.8.12.5 Delegation Facilities

- At CSI level 0, no delegation is supported.
- At CSI level 1, the initiating principal's identity can be delegated to the target. It is either delegated or not - there are no other restrictions on delegation.
- At CSI level 2, the initiating principal's privileges, as well as identity, can be delegated to the target. Delegation can be controlled further, restricting the targets to which the attributes can be delegated. These restrictions must be specified by administrative action, as there are no interfaces specified in to do this in this specification.

Level 2 protocols are also defined which allow support of composite delegation; however, support of this is not required by conformant ORBs.

15.8.12.6 Non-repudiation

Non-repudiation relies on NR credentials for handling NR evidence tokens. The same credentials can be used for secure invocations and non-repudiation. This will only be possible if compatible security technology is used for non-repudiation and secure invocation. While no specific security technology is mandated for non-repudiation, it is expected that this will use public key technology. Common credentials usable for both purposes are expected to use public key technology, to fit with public key mechanisms (SPKM or the CSI-ECMA public key option), rather than with secret key mechanisms.

15.8.12.7 Security Policies

Security policies are potentially sharable between ORBs if they use only identities and privileges which are available at both ORBs and can be transmitted between them. For example, a **DomainAccessPolicy** that uses roles must receive requests from an ORB which can generate them via a CSI level 2 protocol which can transmit roles.

15.8.13 Model for Use and Contents of Credentials

The CORBA Security model includes security functionality enforced during object invocations and by applications, as shown in Figure 15-56 on page 196.

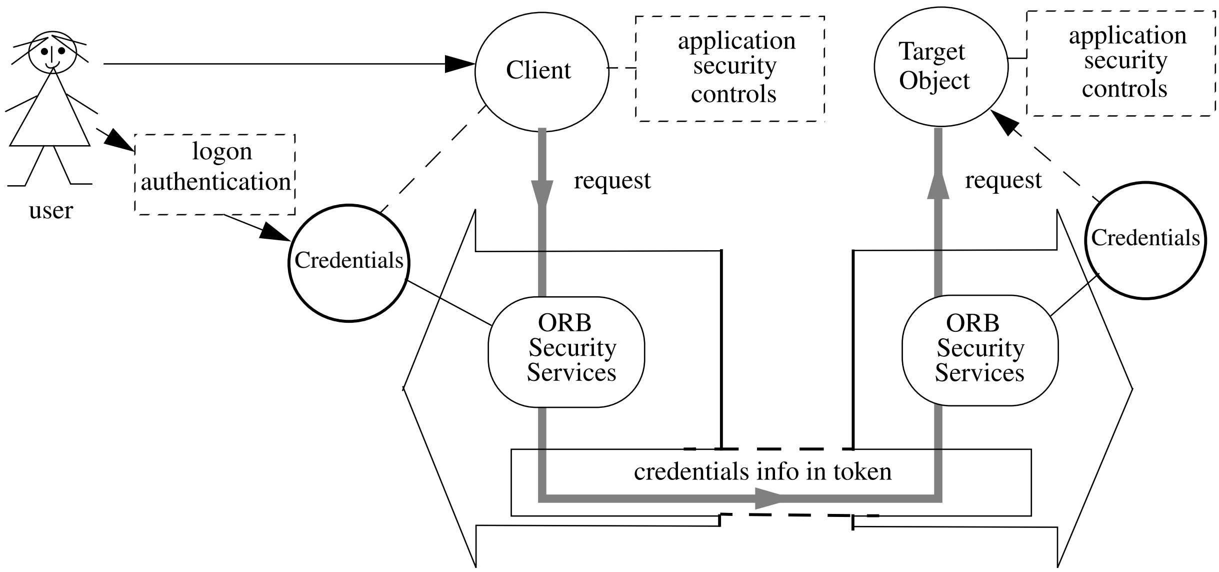


Figure 15-56 Security Functionality Enforced During Object Invocations and Applications

Most of the security services utilize the principal’s credentials either at the client (before invoking the target object) or at the target. For example, the ORB security services use these credentials for secure associations, access control, and auditing.

To fit with the standard CSI security mechanisms, user/principal authentication must produce credentials suitable for both client-side security controls and to fit with the security mechanisms used for secure invocations. A single credential’s object may have security context information for more than one mechanism. Security services at the client application use these credentials to enforce security there.

Access control policies at the target generally depend on the initiating principal’s privilege attributes (which generally includes an identity). Normally they rely on information from the credentials being passed from the client to the target. Other access policies may use the pull model for obtaining privileges at the target. For example, an access policy at the target could obtain the access identity using the **get_attributes** function. It could then call, in a non-standard way, on whatever service provides privileges in this case. Alternatively, an attribute Mapper (see “Attributes at the Target” on page 15-198) could be used before calling the access policy (if this optional facility is supported).

Audit policies generally require an audit id, though this may be derived like the access id from a single identifier.

This specification allows unauthenticated and authenticated users; however, unauthenticated principals do not have identity attributes or privilege attributes. In the protocols defined here, principals must be authenticated.

The privilege and other attributes, as seen by the **AccessDecision** object at the target, may not be those passed from the client because the security mechanism may have moderated what is available to the object system.

15.8.13.1 Credential Content at the Client

Credentials are made available to the client as the result of authenticating the user (or other principal), though they may be modified later. Authenticated users have two types of attributes visible to applications and relevant to secure interoperability:

1. Privilege attributes used for access control. These include the **AccessId** (the principal's identity as used for access control); other standard CORBA security attributes such as **GroupId**, **Role**, **Clearance** and enterprise-defined attributes.
2. Identity attributes used for purposes other than access control. Only the audit identity is relevant here.

   At CSI levels 0 and 1, the only attributes which must be visible to the client and target are the **AccessId** and **AuditId**. These will normally be the user's security name.

   At CSI level 2, a wider range of privilege attributes is supported.

   - All conformant ORBs can generate (via security services) credentials with the following privilege attributes:
     - **AccessId**
     - **AuditId**
     - **Role**
     - **GroupIds** - a primary group and other groups
   - There may be a single identity (e.g., the access identity) which can also be used for auditing, or separate **AccessId** and **AuditId** may be generated. **AuditId** may be anonymous.
   - Optionally, there may also be other privilege attributes including user-defined attributes.

15.8.13.2 Attributes During Transmission

At levels 0 and 1, only the principal's identity is transmitted. No other attributes are transmitted.

At level 2, a wide range of privileges can be transmitted including standard CORBA attributes and optionally user-defined ones. Attributes may have individual defining authorities, as at the IDL interface, or share a defining authority.

15.8.13.3 Attributes at the Target

At CSI levels 0 and 1, when only a single identity (e.g., the security name) is transmitted, that single identity is used to generate the **AccessId** and the **AuditId** at the target. When using the CSI-ECMA protocol at level 0 or 1, principal identity attributes are transmitted separately from the security name; therefore, the **AccessId** and **AuditId** do not have to be generated from the security name.

At CSI level 2, all conformant ORBs can accept:

- Separate access and audit ids or a single identity used for both purposes.
- Transmission of any privileges defined in Appendix Section A.11.1, "Attribute Types," on page 15-307, and any privileges with Object Identifiers which can be mapped to **SecurityAttributes**.

This range of privileges can be used in access decisions at the target. Even if these privileges are not used by the invocation access policy to control access to the target object, they may be obtained by the application using **Current::get_attributes** or **Credentials::get_attribute** and used in application access decisions.

The attributes at the target appear as defined in "Privilege Attributes" on page 15-238. For example, they have:

- an Attribute type (family definer, family, and the type within this family),
- a defining authority, and
- the attribute value.

The attributes may need to be mapped from their form in transit to the form used at the IDL interface in response to **get_attribute** calls. An attribute mapper may be needed, as shown in Figure 15-57.

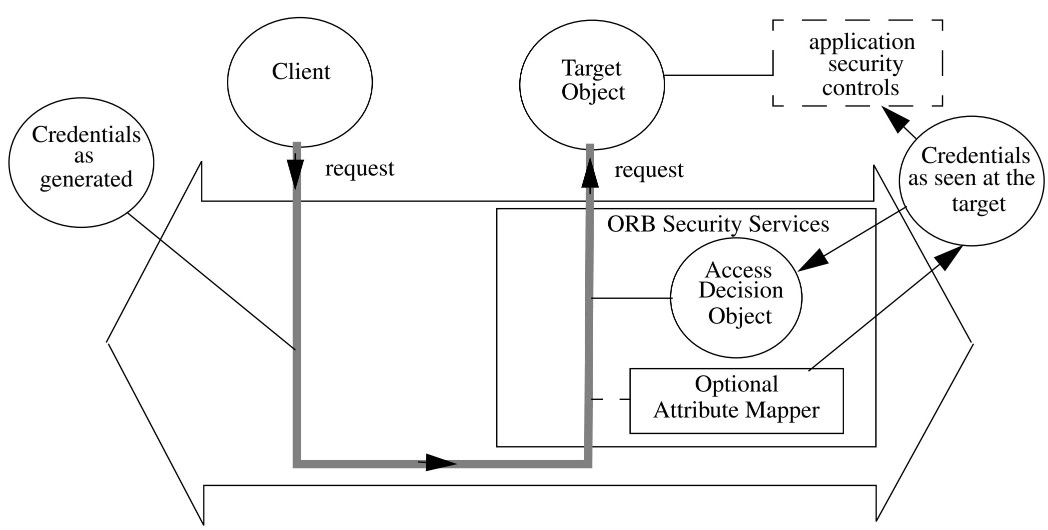


Figure 15-57 Attribute Mapper Diagram

This mapping depends on:

- Which functionality level is supported. At levels 0 and 1, a single name must be mapped to provide both **AccessId** and **AuditId**. This will be the security name if the protocol does not carry a separate **AccessId** or **AuditId**; both the SPKM and GSS-Kerberos protocols use the security name.
- Whether the access control decisions at the target uses attribute values which are valid externally from the ORB/operating system (for example, in a domain of heterogeneous systems), or whether the Access policies use local attributes (such as operating system ids). In line with the OMG requirement for portability, externally valid attributes are the norm, and must be supported in conformant ORBs (so that an application which includes administration of its access policy is portable between unlike systems). Mapping to local attributes may also be provided, but is not standardized in this specification.

15.8.13.4 Mapping Security Names to Externally Valid Identities

Where the only client attribute transmitted is the security name, CSI-conformant ORBs map this onto both the **AccessId** and **AuditId** in the received credentials. These both have the same value.

When using the GSS-Kerberos, the security name protocol has two components: a realm name and a principal name. The security name is of the form ***principal@realm***. The principal name may be a multi-component name with components separated by slash (/) - see [12] section 2.1.1.

When using a public key-based mechanism, the security name is a directory name. This is a multi-part name (e.g., country, organization, organization unit, surname, and common name). The security name is returned from the security mechanism in the form of a string complying with [4] for the string representation of distinguished names. The separators between components of the name may be commas or semicolons.

In both cases, the full Security name is used as the value for the **AccessId** and **AuditId** in the IDL SecurityAttributes. This means the form of these attributes are dependent on the security mechanism used, as Kerberos and X.500 names have different forms.

15.8.13.5 Mapping Other Attributes to Externally Valid IDL Attributes

Other security attributes may also be transmitted from the client when using the CSI-ECMA protocol. For example, at level 2, there could be a **Role**, **GroupId**, and enterprise-specific attributes as well as **AccessId** and/or **AuditId**. Also, separate **AccessId** and **AuditIds** may be transmitted.

In general, these will already have values which are valid outside a particular ORB and operating system; therefore, the mapping is mainly to put these in the form of an IDL SecurityAttribute. However, if a separate **AuditId** has not been transmitted, the **AuditId** value will be copied from the **AccessId**. Also, if a separate defining authority is not transmitted for an attribute, the defining authority for the attribute in IDL is set from the issuer Domain of the authority who generated the Privilege Attribute Certificate containing the privileges. Note also that the target security policy may restrict which of the attributes are available to the application.

Attribute types in transmission are identified by Object Identifiers. For the standard attribute types such as Role or GroupId (as defined in Appendix Section A.11.1, “Attribute Types,” on page 15-307), the type is automatically translated to the appropriate CORBA family and attribute type. The value is also re-encoded, if needed, from ASN.1 to the equivalent IDL type.

We propose that OMG should register itself in the ISO Object Identifier space. A **SecurityAttribute** type where there is a family definer registered with OMG (see Appendix Section A.19, “Values for Standard Data Types,” on page 15-307) can then be transmitted with an Object Identifier of:

<iso>..<omg>.<security>.<family_definer>.<family>.<attribute type>

which then can be mapped automatically onto the CORBA **SecurityAttribute** structure.

Attributes other than the standard attributes and those with CORBA family Object Identifiers are not guaranteed to be understood at the target; therefore, they may not be automatically mapped to CORBA families and types. Such mapping can be done by an optional attribute mapper which understands these attribute types.

15.8.13.6 Mapping to Local Attribute Values

An ORB can support mapping of the security name and other attributes to local operating system values such as UNIX uids and gids. This mapper could generate different **AccessIds** and **AuditIds**. Note that when using local values, the application (particularly the access policy administration) will not be portable to other types of systems.

Mapping of these values is specific to the ORB and/or operating system. This standard does not specify how this mapping is done, whether it calls on other software to do it, or what types of values it generates. However, the defining authority in the IDL SecurityAttribute must identify the local environment responsible for the meanings of these values, so the application can determine where these values are valid.

Mapping to local attributes may be done by an optional attribute mapper (see “Attribute Mapping” on page 15-203).

15.8.14 CORBA Interfaces

In this section:

- Profiles of interfaces defined in sections 15.5 “Application Developer’s Interfaces” through 15.7 “Implementor’s Security Interfaces”,
- Values of certain parameters relevant to these profiles are defined, and

- Restrictions applications that use the Security interfaces must adhere to in order to conform to the CSI standard are defined.

15.8.14.1 Service Options for Common Secure Interoperability

The following Service Options are returned by **ORB::get_service_information** representing the level of CSI that is supported by the ORB:

```
module Security {
    const CORBA::ServiceOption CommonInteroperabilityLevel0 = 10;
    const CORBA::ServiceOption CommonInteroperabilityLevel1 = 11;
    const CORBA::ServiceOption CommonInteroperabilityLevel2 = 12;
};
```

The common interoperability protocols supported are identified using a **ServiceDetail** structure with a **ServiceDetailType** of **Security::SecurityMechanismType**, as described in Section 15.5.2, "Finding Security Features," on page 15-90. The values for the CSI mechanisms are defined in Appendix Section A.10, "General Security Data Module," on page 15-283.

15.8.14.2 Mechanism Types

The mechanism at the application interface is defined as **Security::MechanismType** (a string). CSI mechanisms are encoded in the **MechanismType** string by concatenating a mechanism id and zero, one, or more cryptographic profiles separated by commas.

The mechanisms supported by an object are identified by tags in its IOR. In the MechanismType, the mechanism is identified by a "stringified" form (e.g., the integer value 123 represented as the string "123") of the **TAG_x_SEC_MECH** id value for that mechanism. Mechanisms supported by SECIOP-based protocols are:

- **SPKM_1** or **SPKM_2**: the level 0 public key mechanisms using the SPKM protocol.
- **KerberosV5**: the level 1 secret key mechanism using GSS Kerberos protocol.
- **CSI_ECMA_Secret**: the CSI-ECMA secret key mechanism, using Kerberos V5.
- **CSI_ECMA_Hybrid**: the CSI-ECMA mechanisms which uses secret key technology for key distribution within a domain, but public key between domains.
- **CSI_ECMA_Public**: the CSI-ECMA public key mechanism.

Cryptographic profiles are identified by a "stringified" form of the **CryptographicProfile** value as used in the IOR.

MechanismType is used in a number of operations. These include operations that:

- Deal with the mechanisms and cryptographic profiles in **MechanismsPolicy** object for use with **get_policy** and **set_policy_overrides** on an object reference. In this case, the **mechanisms** attribute of the **MechanismPolicy** object (see "Client-Side Invocation Policy Objects" on page 15-101), contains all the Cryptographic profiles available with that mechanism to communicate with that target.

- Specify a security mechanism to use when talking to a target (e.g., using the **MechanismPolicy** object with the **set_policy_overrides** on an object reference and **Vault::Init_security_context** on the **Vault**). In this case, either just the mechanism name may be specified (in which case, a default cryptographic profile will be used) or a mechanism name and cryptographic profile may be specified.

The **get_service_information** operation on the ORB can also return the mechanism, though in this case, it is in the form of a **sequence<octet>**.

Mechanism tags in the IOR and mechanism type Object Identifiers (as in GSS-API) in SECIOP messages are also used as appropriate.

15.8.14.3 Delegation-Related Interfaces

Interfaces to handle no delegation, simple delegation, and composite delegation (hence delegation interfaces for CSI levels 0, 1, and part of 2) are defined in Section 15.5.10, “Delegation Facilities,” on page 15-120).

CSI level 2 also supports controls on the delegation of credentials. How to specify these controls is not included in this specification. It is assumed that it is handled by administrative action. For example, it may be done by associating the delegation controls with a user or an attribute set selected when the user logs on or selects attributes at other times. Management of attributes associated with a principal is considered out of this specification’s scope.

No facilities are currently defined for an application object to specify controls it wishes to apply on delegating its credentials. In the future, such facilities may be considered for CORBA Security - see Appendix Section F.13, “Advanced Delegation Features,” on page 15-362.

15.8.15 Support for CORBA Security Facilities and Extensibility

This CSI specification assumes that the ORB conforms to at least CORBA Security mandatory facilities (except for delegation at CSI level 0), and requires that this functionality can be supported across different ORBs using any of the CSI levels specified here.

The CORBA Security specification allows use of a wide range of security policies, facilities, and mechanisms. Conformant ORBs can restrict which of these can be used during interoperability, as follows:

- The protocol may not carry the privileges the target needs for some of its access policies. For example, at CSI levels 0 and 1 only an identity is supported.
- It may not carry the type of audit identity needed for the audit policy. For example, it may not be able to carry an anonymous **AuditId**.
- It may not support composite delegation. (CSI levels 0 and 1 do not; in CSI level 2 it is not mandatory).
- There are restrictions on the SECIOP exchanges (e.g., separate request and response protection is not supported).

- Unauthenticated users may not be supported (All CSI levels).

15.8.16 Security Replaceability for ORB Security Implementors

Security policy implementations could be replaced to provide new security policies as discussed in Section 15.7.3, “Replaceable Security Services,” on page 15-170.

This common Interoperability specification affects replaceability in two areas:

1. Mapping of attributes as described in Section 15.8.13, “Model for Use and Contents of Credentials,” on page 15-196 affects replaceable security policies which use these attributes.
2. Use of the Generic Security Services API (GSS-API) within the **Vault** and **Security Context** implementation objects described in Section 15.7.2, “Implementation-Level Security Object Interfaces,” on page 15-161, should make these objects independent of the particular security mechanisms used.

15.8.16.1 Attribute Mapping

As described in “Attributes at the Target” on page 15-198, the form of attributes may need to be mapped before being made available to a target security policy (**AccessPolicy** or **AuditPolicy**) or to the target object.

No interface for an attribute mapper is currently defined; therefore, it is not possible to replace attribute mapping independently of the ORB/security mechanism. Such an interface may be defined in the future.

15.8.16.2 Use of GSS-API

The choice of security mechanism is not visible outside the **Vault** and **Security Context** objects, except for the identification of the **Mechanism** (and associated cryptographic profiles) in the IOR and in the **MechanismPolicy** object (see “Client-Side Invocation Policy Objects” on page 15-101).

The **Vault** and **Security Context** can use GSS-API to implement their security functions, and so remain independent of security mechanisms.

If only CSI level 0 or 1 facilities are used, the standard GSS-API interface (as defined in RFC 1508) can be used. If CSI level 2 facilities are needed, this requires use of attributes other than the security name, and may also use delegation controls. Therefore, it requires use of an extended GSS-API, such as [12].

Use of GSS-API is a recommendation, but is not proposed as a conformance option for this CSI specification or for the CORBA Security specification.

15.9 Secure Inter-ORB Protocol (SECIOP)

To provide a flexible means of securing interoperability between ORBs, a new protocol is introduced into the CORBA Interoperability Architecture. This protocol sits below the GIOP protocol and provides a means of transmitting GIOP messages (or message fragments) securely.

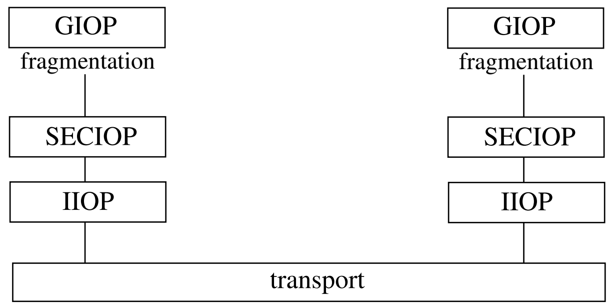


Figure 15-58 Position of SECIOP Protocol

SECIOP messages support the establishment of Security Context objects and protected message passing. Independence from GIOP allows the GIOP protocol to be revised independently of SECIOP (e.g., to support request fragmentation). A synchronized pair of **Security Context** objects and their corresponding sequencing state is called a *security association*.

SECIOP is sub-layered into a Sequencing Layer and Context Management Layer.

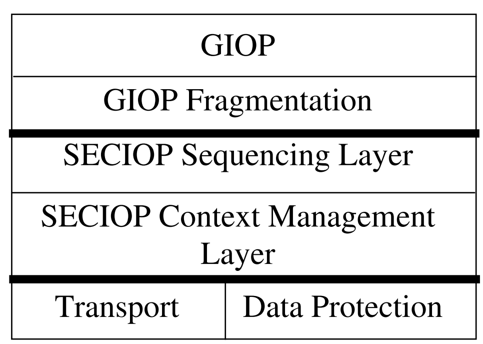


Figure 15-59 Sublayers of SECIOP

This specification assumes that SECIOP provides services to the GIOP Fragmentation Layer. Providing the interface to GIOP fragmentation is the SECIOP Sequencing Layer. It has the responsibility of securely and reliably delivering GIOP fragments to the correspondent. It encapsulates GIOP fragments into frames for protection by the SECIOP Context Management Layer. It also uses frames that do not carry fragments to coordinate the distributed sequence number state bound to the security association. SECIOP frames are encoded in CDR and delivered to the SECIOP Context Management Layer.

The SECIOP Context Management Layer accepts frames from the Sequencing layer and encapsulates them in a Context Management message. These messages are cryptographically protected by tokens, which are the product of the Data Protection layer, normally GSSAPI. The Context Management Layer carries Data Protection tokens in SECIOP messages for the purpose of both managing security associations and for securing frames moving between it and the correspondent. The Context Management layer uses the Transport layer to communicate with the correspondent. The Context Management layer is driven by the finite state machine defined in Table 15-13 on page 15-218 and Table 15-14 on page 15-221.

15.9.1 Architectural Assumptions

SECIOP is designed to support a rich variety of different software implementation architectures. In order to operate in the most sophisticated of these, the design assumes both clients and targets are multi-threaded and that a single TCP connection can support multiple security associations.

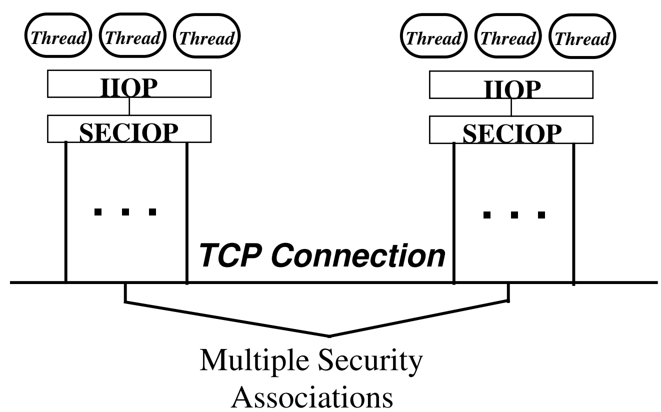


Figure 15-60 Architectural Assumptions

This specification assumes the following environmental and implementation characteristics:

- Each SECIOP-secure association is bound to a single transport connection. This ensures that GIOP fragments are not reordered due to thread scheduling anomalies. It also guarantees that a response to a GIOP request returns on the same transport connection as the request, which is required by the GIOP specification.
- SECIOP may use multiple security associations over the same transport connection. This allows implementations to multiplex SECIOP traffic, which can improve performance.
- SECIOP ensures that fragments are sent over transport connections in their sequence number order. This means that once an SECIOP sequence number is assigned to a fragment, the fragment will be processed by the Data Protection layer and sent over transport before any other fragment with a larger sequence number protected by the same security association.

- When a transport connection is closed, all SECIOP-secure associations using it are closed as well. This may require discarding fragments on the Sequencing layer retransmission queue that have not yet been acknowledged. This is acceptable, since closing a transport connection forces GIOP to mark any outstanding Requests as MAYBE. Furthermore, closing a transport connection must be visible to both sides of the connection, so both sides of the security association will follow this rule.
- There is always a listener at the client and server prepared to receive and process SECIOP messages. This is necessary, since the loss of security context information by one side or the other requires a re-establishment of the security association. This in turn requires both client and server to be listening for security context management messages.
- Both the client and server may initiate security context establishment (i.e., send the **EstablishContext** message). This is necessary when a server needs to return a response to the client but discovers that the security association is no longer valid (e.g., it has timed out).
- SECIOP sequence numbers should never wrap around to zero. If they did, it would introduce a replay threat. Consequently, when the SECIOP Sequencing Layer receives an acknowledgment to a fragment with a sequence number equal to 1/2 the precision of an unsigned long (the type used for sequence numbers), it must discard the existing security association and establish a new one. This rule derives from the sequencing algorithm property that up to 1/2 of the possible sequence numbers in the higher 1/2 of the sequence number space may be used for new fragments before the fragment associated with the last sequence number in the lower 1/2 of the space is acknowledged. Note that the SECIOP sequencing state should not be discarded when a new security context is established.
- There is Data Protection protocol information (e.g., GSSAPI tokens) carried within SECIOP messages. This protocol should be configured so it does not itself provide sequencing services. Otherwise, there could be interference between the two layers, causing unnecessary lost service.

15.9.2 SECIOP Sequencing Layer

SECIOP sequencing uses a modified data link layer protocol based on one in production at Lawrence Livermore National Laboratory for over 10 years. This protocol, called modified ALP, is described below.

SECIOP Sequencing layer frames are carried in **MessageInContext** messages (see “Message Definitions” on page 15-214). The **message_protection_token** in this message is defined to be an opaque sequence of octets. In order to support sequencing, however, the Sequencing layer defines the structure of these octets as follows (the definition of **MessageInContext** is repeated here for completeness):

```
struct MessageInContext {
    ContextIdDefn       message_context_id_defn;
    TokenType           message_protection_token_type;
    ContextId           message_context_id;
    sequence<octet>     message_protection_token;
```

```
};
```

message_protection_token is obtained by processing the frame header encoded in CDR as a **SequencingHeader followed by the octets of the frame data.** The combination of frame header and frame data is called a **SequencedDataFrame**.

The **frame_header** field is always present in a **SequencedDataFrame**; however, the **frame_data** field may or may not be present. If not present, the length of the **MessageInContext** message includes only the octets up to and including the frame header.

The **SequencingHeader** has the following definition:

```
struct SequencingHeader {
    octet           control_state;
    unsigned long   direct_sequence_number;
    unsigned long   reverse_sequence_number;
    unsigned long   reverse_window;
};
```

The **control_state** field contains information necessary for the reliable delivery of frame data between the correspondents. It is encoded as follows (**control_state[x]** is bit **x** in the octet, where bit **0** is the least significant bit):

```
control_state[0] : direct_phase
    control_state[1] : direct_fragment
    control_state[2] : direct_reply
    control_state[3] : reverse_phase
```

15.9.2.1 Protocol State

The new version of SECIOP uses a variant of ALP (A Link Protocol) a data link layer protocol. Its design relies on the principal of state-exchange, a coherent design strategy that produces protocols that are easy to understand, clearly documented, and lend themselves to rigorous analysis.

It is assumed that the reader is familiar with this link-layer protocol. Those unfamiliar with it are referred to the paper [18].

The main body of this paper establishes the rationale for the state-exchange model, while Appendix A documents the ALP protocol itself.

To embed ALP within SECIOP, each participant in a security association maintain the state used for sequencing. This state is embodied in several variables that the participant manages as well as a queue of data fragments.

These are:

output_queue	A queue of fragments. SECIOP is responsible for securely and reliably moving them to the correspondent.
output_phase	A boolean indicating a stream of transmissions.
output_sequence_number	The sequence number associated with the oldest fragment on **output_queue**.
output_count	The number of fragments in **output_queue** that have been transmitted but not yet accepted or rejected.
output_window	The window size for output fragments.
output_length	The length of the **output_queue**.
input_phase	The phase expected with the next input fragment.
input_sequence_number	The sequence number expected for the next fragment.
input_window	The window size for input fragments.
input_reply	A boolean, which if set indicates at least one frame should be sent.

In addition to these variables, the SECIOP Sequencing layer has the following functions and procedures available:

receive*()*	Returns a received frame.
newframe*()*	Returns an empty frame buffer (i.e., a **SequencedDataFrame** struct).
send(f)	Sends the frame **f**.
discard(f	Discards the frame **f**.
pop(q*)*	Removes and discards the leading element **q[0]** of the queue **q**. The index of the remaining elements is decremented by one.
forward(d*)*	Forwards the fragment **d** to the GIOP fragmentation layer.
mod(n,m)	Returns the remainder from the division of the integer **n** by the positive integer **m**.
min(n_1,n_2,..., n_x)	Returns the smallest of the integers $\mathbf{n_1}$ through $\mathbf{n_x}$.
resync()	Signals the SECIOP Context Management layer to discard the old security association bound to the sequencing state and establish a new one. [NB: this is not included in the original ALP definition, since the notion of a security context is not germane to its original purpose].
frame_data(f)	The **frame_data** field of a **SequencedDataFrame** message **f**.

Finally, the value **M** is defined to be the number of values that can be carried by an **unsigned long**.

15.9.2.2 Protocol Initialization

The next three sections describe the operation of the Sequencing layer. The algorithms are expressed in a pseudo-ALGOL syntax (with slight modifications from the C programming language to facilitate writing conditional expressions).

When the GIOP fragmentation layer requests the transport of a fragment to a destination for which no SECIOP-secure association exists, the SECIOP Sequencing layer creates a state record consisting of the variables defined in the last section and initializes them as follows:

output_queue := empty;

output_phase := 0;

output_sequence_number := 0;

output_count := 0;

output_window := 0;

output_length := 0;

input_phase := 0;

input_sequence_number := 0;

input_window := [an implementation defined value < M/2];

input_reply := 1;

In the original definition of ALP, the initial values of some of these variables was unspecified. This specification defines these initial values so that there need be no handshaking activity between the correspondent's SECIOP Sequencing layer code in order to move the first fragment. This facilitates transaction style operations in which a security association is established without mutual authentication, thus allowing the first fragment to be sent without waiting for an SECIOP reply.

Another slight change from the original definition of ALP is the requirement that the window size must never be set greater than (M/2)-1. This restriction is necessary so that two acknowledgments carrying equal sequence numbers referring to different fragments are never protected using the same security context. Without this restriction, there is a hazard that an intruder could replay an acknowledgment to a fragment not received, thereby causing the fragment to be dropped.

Once a security context is established, the SECIOP Sequencing layer processes the information in a SequencedDataFrame according to the algorithms given in the next two sections.

15.9.2.3 Upon Receipt of a SequencedDataFrame

Note – This text is taken directly from the cited paper and slightly modified to adapt it to using security contexts. The code that has been modified is called out by a solid black line on the left side.

The receiver code below is called on both the target and client sides when the SECIOP Finite State Machine (FSM) is in state S3 and a **MessageInContext** arrives.

```
begin comment This algorithm should be executed after receipt of each non-
                    erroneous frame;
    f := receive();
    if direct_sequence_number(f) == input_sequence_number
        and
     direct_phase(f) == input_phase
    then
        if direct_fragment(f) == 1 and input_window > 0
        then comment An input fragment has arrived in sequence. Accept it;
            input_sequence_number := mod(input_sequence_number + 1, M);
            forward(frame_data(f));
        fi;
    else comment An input fragment has been lost. Prepare to accept retransmissions;
        input_phase := 1 - direct_phase(f);
    fi;
    if mod(reverse_sequence_number(f)-output_sequence_number, M) <= output_count
    then comment The received reverse sequence number is not anomalous;
        while reverse_sequence_number(f) != output_sequence_number
            do comment Discard accepted output fragments;
                pop(output_queue);
                output_sequence_number := mod(output_sequence_number + 1, M);
                if mod(output_sequence_number, M/2) == 0
                then comment all fragments up to and including (M/2)-1 have been
                    acknowledged. Use a new security context for future fragments
                    to avoid replays. Resynchronizing the security context when
                    exactly half of the sequence number space has been “used”
                    achieves two objectives : 1) it ensures that no two fragments with
                    the same sequence number are protected by the same security
                    context, and 2) it ensures that two acknowledgments carrying the
                    same sequence  number, but acknowledging different fragments
                    are not protected using the same security context. The latter
```

objective requires the further limitation that the window size is never set greater than (M/2)-1;

```
                resync();
                fi;
            output_count := output_count - 1;
            output_length := output_length - 1;
            od;
        output_window := reverse_window(f);
    fi;

    if reverse_phase(f) != output_phase
    then comment Prepare to retransmit rejected output packets;
        output_phase := reverse_phase(f);
        output_sequence_number := reverse_sequence_number;
        output_count := 0;
    fi;

    if direct_reply(f) == 1 or output_length > 0
    then comment State is unsatisfactory;
        input_reply := 1;
    fi;
    discard(f)
    end;
```

15.9.2.4 Sending a SequencedDataFrame

This sending code is called on the target and client side when the Sequencing Layer caller has a fragment to send. Certain events within the Sequencing layer also cause this algorithm to be executed. Specifically, the sending algorithm is executed when the receiving code in the previous section is executed and a non-erroneous frame is received. Also, **input_reply** should be set to 1 and the sending code executed:

1. when an erroneous frame is received;
2. when a new security context is established;
3. when an **EstablishContext** message is sent with messages allowed;
4. when **input_window** is changed by the implementation; and
5. upon initialization of the Sequencing state.

```
    begin
```

```
while output_count < min(output_window, output_length) or input_reply == 1
    do comment A frame should be sent;
        f := newframe();
        input_reply := 0;
        direct_phase(f) := output_phase;
        direct_sequence_number(f) := mod(output_sequence_number +
                                        output_count, M);
        if output_count < min(output_window, output_length)
        then comment A fragment could be included in the frame.
            direct_fragment(f) := 1;
            frame_data(f) := output_queue[output_count];
            output_count := output_count + 1;
        fi;
        if output_length > 0)
        then comment Not all packets have as yet been accepted;
            direct_reply(f) := 1
        fi;
        reverse_phase(f) := input_phase;
        reverse_sequence_number(f) := output_sequence_number;
        reverse_window(f) := output_window;
        send(f);
    od
end
```

15.9.3 SECIOP Context Management Layer

The SECIOP Context Management Layer establishes and controls a secure association between a client and target. It also provides a means for the transmission of protected messages between clients and targets.

15.9.3.1 SECIOP Context Management Layer Message Header

SECIOP Context Management messages share a common header format with GIOP messages defined in the *Common Object Request Broker: Architecture and Specification*. The fields of this header have the following definition:

- **magic** - identifies the protocol of the message. Each protocol (GIOP, SECIOP) is allocated a unique identifier by the OMG. The value for SECIOP is “SECP.”
- **protocol_version** - this contains the major and minor protocol versions of the protocol identified by magic. The value for the version of SECIOP defined here is 1 major version, 1 minor version. This field is called **GIOP_version in GIOP::MessageHeader_1_1**.

- **byte_order** - as in the GIOP header definition.
- **message_type** - this is the protocol-specific identifier for the message.
- **message_size** - as in the GIOP header definition.

15.9.3.2 SECIOP Context Management Layer Protocol

Where possible, SECIOP Context Management messages are sent with GIOP messages rather than as separate exchanges. However, this is not always possible (e.g., when the client wishes to authenticate the target before it is prepared to send a GIOP message).

The SECIOP Context Management Layer has the following message types:

```
module SECIOP
    enum MsgType {
        MTEstablishContext, MTCompleteEstablishContext,
        MTContinueEstablishContext, MTDiscardContext,
        MTMessageError, MTMessageInContext
    };

    typedef unsigned long long ContextId;

    enum ContextIdDefn {
            CIDClient,
            CIDPeer,
            CIDSender
    };

    enum ContextTokenType {
            SecTokenTypeWrap,
            SecTokenTypeMIC
    };
};
```

15.9.3.3 ContextId

This type is used to define the identifiers allocated by the client and target for the association.

15.9.3.4 ContextIdDefn

This enum is used to define the kind of context identifier held in an SECIOP message. The context identifier will either be the one specified by the client which established the context or it will be the identifier associated with the receiver of the message (i.e., the request target for request or request fragment messages, or the request client for reply or reply fragment messages). The value must equal Client if the value of **target_context_id_valid** in the **CompleteEstablishContext** was false or the message has not yet been exchanged. It must equal Peer if the value of **target_context_id_valid** in the **CompleteEstablishContext** was true. The use of peer identifiers allows the recipient of the message to more efficiently find its security context. The values are defined as:

- **CIDClient** - the context id is that of the association's client.
- **CIDPeer** - the context id is that of the recipient of the message.
- **CIDSender** - the context id is that of the sender of the message. This is only used with the **DiscardContext** message when the sender of the **DiscardContext** message has no context and has received a message which it cannot process.

15.9.3.5 *TokenType*

This type is used to indicate the type of **message_protection_token** carried by a **MessageInContext** message. The value **SecTokenTypeWrap** indicates the token was returned by a **GSS_Wrap()** call, while the value **SecTokenTypeMIC** indicates the token was returned by a **GSS_GetMIC()** call.

15.9.3.6 *Message Definitions*

EstablishContext

This message is passed by the client to the target when a new association is to be established. Its definition is:

```
struct EstablishContext {
    ContextId           client_context_id;
    sequence <octet>    initial_context_token;
};
```

- **client_context_id** - this is the client's identifier for the security association. It is passed by the target to the client with subsequent messages within the association. It enables the client to link the message with the appropriate security context.
- **initial_context_token** - this is the token required by the target to establish the security association. It contains a mechanism version number, mech type identifier and mechanism-specific information required by the target to establish the context. It may be sent with a protected message (for example if the client does not wish to authenticate the target).

CompleteEstablishContext

This message is returned by the target to indicate that the association has been established. It is sent as a reply to an establish context or continue establish context. It may be sent with a GIOP reply or reply fragment. Its definition is:

```
struct CompleteEstablishContext {
    ContextId           client_context_id;
    boolean             target_context_id_valid;
    ContextId           target_context_id;
    sequence <octet>    final_context_token;
```

```
};
```

- **client_context_id** - this is the client's identifier for the security association. It is returned by the target to the client to enable the client to link the message with the appropriate security context.
- **target_context_id_valid** - this indicates whether the target has supplied a **target_context_id** for use by the client. True indicates that the following field is valid.
- **target_context_id** - the targets identifier for the association. It is passed by the client to the target with subsequent messages. It enables the target to associate a local identifier with the context to allow the target to identify the context efficiently.
- **final_context_token** - this is the token required by the client to complete the establishment of the security association. It may be zero length.

ContinueEstablishContext

This message is used by the client or target during context establishment to pass further messages to its peer as part of establishing the context. It may be the response to an establish context or to another continue establish context. It is defined as:

```
struct ContinueEstablishContext {
    ContextId           client_context_id;
    sequence <octet>    continuation_context_token;
};
```

- **client_context_id** - the client's identifier for the association. It is used by both client and target to identify the association during the establishment sequence.
- **continuation_context_token** - this is the security information required to continue establishment of the security association.

DiscardContext

This message is used to indicate to the receiver that the sender of the message has discarded the identified context. Once the message has been sent the sender will not send further messages within the context. The message is used as a hint to enable contexts to be closed neatly. Its definition is:

```
struct DiscardContext {
    ContextIdDefn       message_context_id_defn;
    ContextId           message_context_id;
    sequence <octet>    discard_context_token;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.
- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies.
- **discard_context_token** - optional token provided by the sender to assist the receiver in cleaning up its security context state.

MessageError

This message is used to indicate an error detected in attempting to establish an association either due to a message protocol error or a context creation error. The message is also used to indicate errors in use of the context.

```
struct MessageError {
    ContextIdDefn       message_context_id_defn;
    ContextId           message_context_id;
    long                major_status;
    long                minor_status;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.
- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies. It is either the client's identifier for the context (type client) or the receiver of the messages identifier (type peer).
- **major_status** - the reason for rejecting the context. The values used are those defined by the GSS API (RFC 1508) for fatal error codes.
- **minor_status** - this field allows mechanism-specific error status to further define the reason for rejecting the context. It is not defined further here.

MessageInContext

Once established, messages are sent within the context using the **MessageInContext** message. Its definition is:

```
struct MessageInContext {
    ContextIdDefn       message_context_id_defn;
    TokenType           message_protection_token_type;
    ContextId           message_context_id;
    sequence <octet>    message_protection_token;
};
```

- **message_context_id_defn** - the type of context identifier supplied in the **message_context_id** field.
- **message_protection_token_type** - indicates whether the **message_protection_token** is a **SecTokenTypeWrap** or **SecTokenTypeMIC** token.
- **message_context_id** - the context identifier to be used by the recipient of the message to identify the context to which the message applies.
- **message_protection_token** - the sign or seal token for the message. This is a self-defining token which indicates how the message is protected. If the message is not protected the token will be zero length.

For signed and unprotected messages, the **MessageInContext** message is followed by the higher-level protocol message being transmitted within a security context (i.e., GIOP message or message fragment). The length of the higher-level protocol message is included in the length of the **MessageInContext** message. For sealed messages the length of the higher-level protocol message is zero.

15.9.4 SECIOP Context Management Finite State Machine Tables

Table 15-13 on page 15-218 and Table 15-14 on page 15-221 present the state transition rules for the Context Management Layer of SECIOP. The state transitions given in these tables are intended to operate in an environment satisfying the following assumptions:

- Each FSM is associated with a unique pair of principals. When an SECIOP message arrives it is delivered to the FSM associated with the principal from which the message was sent and to which the message is delivered.
- There always exists a sequencing state machine (SSM) in the initialized state with an FSM in state 0 at each end of a TCP connection for those principal pairs without an active SSM/FSM.
- Each SSM is associated with exactly one FSM at a time, although an SSM may be associated with multiple FSMs during its lifetime.
- Each TCP connection can be associated with multiple SSMs.
- Each FSM is associated with exactly one **ContextId** during its lifetime.

15.9.4.1 SECIOP Context Management Protocol State Tables

Note that some mechanisms may start in state S3.

Table 15-13 SECIOP Context Management Finite State Machine -Table 1

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
EstablishContext arrives	**If** create context = OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else if** create context = OK & context incomplete. Send ContinueEstablishContext. S2. **Else** Send MessageError. Terminate SSM. Terminate.	[Target sent EstablishContext at same time Client did. Client's has precedence] S1.	[Target sent EstablishContext at same time Client did. Client's has precedence] S2.	[Target discarded context without telling client] Create a new FSM in state S0. Deliver EstablishContext message to it. Terminate.
CompleteEstablishContext arrives	[A CompleteEstablishContext arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	Complete context with target's context id. **If** OK, S3. **Else**, send MessageError. Terminate SSM. Terminate	Complete context with target's context id. **If** OK, input_reply := 1. Execute send algorithm. S3. **Else**, send MessageError. Terminate SSM. Terminate	[A CompleteEstablishContext arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate

Table 15-13 SECIOP Context Management Finite State Machine -Table 1 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
ContinueEstablish-Context arrives	[A ContinueEstablish-Context arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[A ContinueEstab-lishContext arriv-ing in S1 is illegal] Send MessageEr-ror. Terminate SSM. Terminate	update context state. **If** OK & context complete, Send CompleteEs-tablishContext. input_reply := 1. Execute send algo-rithm. S3. **Else If** OK & con-text incomplete, Send ContinueEs-tablishContext. S2. **Else**, Send MessageEr-ror. Terminate SSM. Terminate	[A ContinueEstablish-Context arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate
MessageError arrives	[A MessageError arriv-ing in S0 is illegal] Terminate SSM. Terminate	Terminate SSM. Terminate	Terminate SSM. Terminate	[target had trouble using its security context and couldn't reestablish it] Terminate SSM. Terminate.
Send Frame [Normal send case.]	**If** create context = OK, Send EstablishCon-text message. **If** Message allowed, Send the frame. S1. **Else** S2. **Else** Terminate SSM. Terminate	Send the frame. S1.	S2.	**If** context valid, Send the frame. S3. **Else** Create a new FSM in state S0. Attach it to SSM. Deliver SendFrame to FSM Terminate

Table 15-13 SECIOP Context Management Finite State Machine -Table 1 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
MessageInContext arrives [Normal receive case.]	[Client has discarded context, but target doesn't know it.] Send DiscardContext. S0	[MessageInContext arriving in state S1 is illegal]] Send MessageError. Terminate SSM. Terminate	[MessageInContext arriving in state S2 is illegal]] Send MessageError. Terminate SSM. Terminate	**If** message OK, Execute receive algorithm. **Else If** context timed out, Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate. **Else If** message bad, but context OK, drop message. input_reply := 1. Execute send algorithm. **Else** Send MessageError. Terminate SSM. Terminate.
DiscardContext arrives	[ignore] S0	[Target doesn't want to create a security association] Terminate SSM. Terminate	[Target doesn't want to create a security association] Terminate SSM. Terminate	[target's context is no longer valid] Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate.
Resync Requested	[ignore. Resync will occur on next Send-Frame request] S0	Terminate SSM. Terminate	Terminate SSM. Terminate	Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. Execute send algorithm. Terminate.

Table 15-14 SECIOP Context Management Finite State Machine - Table 2

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
EstablishContext arrives	**If** create context = OK & context complete, Send CompleteEstablishContext. input_reply := 1. Execute send algorithm. S3. **Else if** create context = OK & context incomplete, Send ContinueEstablishContext. S2. **Else** Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	[Client wants to start over. Always allow this.] discard partial context. Create a new FSM in state S0. Deliver EstablishContext frame to it. Terminate.	[Client discarded context without telling target.] Create a new FSM in state S0. Deliver EstablishContext frame to it. Terminate.
CompleteEstablishContext arrives	[A CompleteEstablishContext arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	Complete context with context id. **If** OK, input_reply := 1. Execute send algorithm. S3. **Else**, send MessageError. Terminate SSM. Terminate	[A CompleteEstablishContext arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate

Table 15-14 SECIOP Context Management Finite State Machine - Table 2 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
ContinueEstablish-Context arrives	A ContinueEstablish-Context arriving in S0 is illegal] Send MessageError. Terminate SSM. Terminate	[illegal state at Target Side]	update context state. **If** OK & context complete, Send CompleteEs-tablishContext. input_reply := 1. Execute send algo-rithm. S3. **Else If** OK & con-text incomplete, Send ContinueEs-tablishContext. S2. **Else**, Send MessageEr-ror. Terminate SSM. Terminate	[A ContinueEstablish-Context arriving in S3 is illegal] Send MessageError. Terminate SSM. Terminate
MessageError arrives	[A MessageErrort arriv-ing in S0 is illegal] Terminate SSM. Terminate	[illegal state at Target Side]	Terminate SSM. Terminate	[target had trouble using its security con-text and couldn't rees-tablish it] Terminate SSM. Terminate.
Send Frame [Normal send case.]	**If** create context = OK, Send EstablishContext message. S2. **Else** Terminate SSM. Terminate	[illegal state at Target Side]	S2.	**If** context valid Send the frame (if not already sent). S3. **Else** Create a new FSM in state S0. Attach it to SSM. Deliver SendFrame to FSM Terminate.

Table 15-14 SECIOP Context Management Finite State Machine - Table 2 *(Continued)*

Event	**No Association (S0)**	**Association being created, message allowed (S1)**	**Association being created, message not allowed (S2)**	**Association exists (S3)**
MessageInContext arrives [Normal receive case.]	[Target has discarded context, but client doesn't know it.] Send DiscardContext. S0	[illegal state at Target Side]	[MessageInContext arriving in state S2 is illegal]] Send MessageError. Terminate SSM. Terminate	**If** message OK, Execute receive algorithm. **Else If** context timed out, Send DiscardContext. Create a new FSM in state S0 Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate **Else If** message bad, but context OK, drop message. input_reply := 1. Execute send algorithm. **Else** Send MessageError. Terminate SSM. Terminate
DiscardContext arrives	[ignore] S0	[illegal state at Target Side]	[Client doesn't want to create a security association] Terminate SSM. Terminate	[client's context is no longer valid.] Create a new FSM in state S0. Attach it to SSM. input_reply := 1. Execute send algorithm. Terminate.
Resync Requested	[ignore. Resync will occur on next SendFrame request] S0	[illegal state at Target Side]	Terminate SSM. Terminate	Send DiscardContext. Create a new FSM in state S0. Attach it to SSM. Execute send algorithm. Terminate.

15.10 The SECIOP-Hosted CSI Protocols

All the SECIOP hosted Common Secure Interoperable (CSI) protocols and mechanisms use common elements as far as possible.

- All mechanisms use IOR tags of the form **TAG_x_SEC_MECH** as defined in "Security Components of the IOR" on page 15-179.
- The component data structure associated with these tags is common for all protocols and mechanisms in this specification.
- Cryptographic profiles are defined in all cases which allow use of relevant algorithms for confidentiality, integrity, etc. Different mechanisms support some of the same algorithms and one way functions.
- The MechanismType as seen at the IDL interface also reflect the mechanism ids and cryptographic profile values in the IOR tags.
- Privilege attributes when CSI level 2 is used are the same whether a secret or public key mechanism is used.
- The basic SECIOP token format and some details (such as token types and ids) are common for all protocols.
- All tag components must be encapsulated using CDR encoding.

These protocols are designed to allow use of GSS-API mechanisms. Use of level 2 facilities such as handling of privileges, as defined in Appendix Section A.19, "Values for Standard Data Types," on page 15-307, imply the use of an extended GSS-API such as [23].

15.10.1 IOR

The IOR **TAG_INTERNET_IOP** profile contains the security tags needed for common secure interoperability using GIOP/IIOP. These security tags may be shared with other (non-IIOP) protocols, including DCE-CIOP.

The security tags describe what the security target supports and requires, and any mechanism-specific data required for secure interoperability using this mechanism.

For common secure interoperability and for all CSI mechanisms and protocols, the IOR must contain at least one appropriate **TAG_x_SEC_MECH** tag.

The IOR may also contain the following tags, as defined in "Security Components of the IOR" on page 15-179:

- **TAG_SEC_NAME** provides the security name and may be shared between mechanisms which use the same form of name. Conformant implementation must be able to accept security names shared between such mechanisms.
- **TAG_ASSOCIATION_OPTIONS** may be shared between mechanisms.
- **TAG_GENERIC_SEC_MECH** whose component definition includes a **sequence <TaggedComponents>** includes a **security_mechanism_type** and can include a security name and association options.

If a mechanism is selected for use, and has a defined security name and/or association option, these values are used in preference to any values defined at the higher level. If no name or association options are defined for the mechanism, then the values of these tags in the IIOP profile are used.

15.10.2 Mechanism Tags

The **TAG_x_SEC_MECH** tags for all the CSI mechanisms defined in this specification have an associated component data structure of the same form:

```
struct <mechanism name> {
    AssociationOptions                  target_supports;
    AssociationOptions                  target_requires;
    sequence <CryptographicProfile> crypto_profiles;
    sequence <octet>                    security_name
};
```

Names for the CSI mechanisms are:

SPKM_1
SPKM_2
KerberosV5
CSI_ECMA_Secret
CSI_ECMA_Hybrid
CSI_ECMA_Public

Tag ids for the mechanisms are:

TAG_SPKM_1_SEC_MECH
TAG_SPKM_2_SEC_MECH
TAG_KerberosV5_SEC_MECH
TAG_CSI_ECMA_Secret_SEC_MECH
TAG_CSI_ECMA_Hybrid_SEC_MECH
TAG_CSI_ECMA_Public_SEC_MECH

- The association options required/supported by the target are defined in Section 15.10.3, “Association Options,” on page 15-225.
- The sequence of **crypto_profiles** defines one or more cryptographic profiles supported by this target using this mechanism as defined in Section 15.10.4, “Cryptographic Profiles,” on page 15-226.
- The security name is defined in Section 15.10.5, “Security Name,” on page 15-227.

15.10.3 Association Options

With all CSI protocols and mechanisms, a secure ORB supporting a target object must be able to put in the IOR any or all of the association options defined in the CORBA Security specification, as required by the target.

All compliant secure ORBs supporting clients must be able to accept all the **target_supports** and **target_requires** association options, and act on these correctly, as defined in "TAG_ASSOCIATION_OPTIONS" on page 15-180.

Two of the association options are replay and misordering detection. While all the protocols in this specification include facilities to detect replay and misordering, in a multi-threading CORBA environment, the calls on the security mechanism are not guaranteed to be made in the same order that the messages they are protecting are transmitted. The facilities in the security mechanisms cannot guarantee that they will correctly detect replay and misordering. An extension to SECIOP is expected in the future to provide these checks. Until this change to SECIOP has been specified and adopted (although these association options may be set) replay and misordering detection is not a mandatory part of this specification.

If no association options are specified in the IOR, a CSI-defined default is assumed.

15.10.4 Cryptographic Profiles

Cryptographic algorithms are used for

- integrity and confidentiality protection of messages,
- establishing the security association between client and target (including peer authentication and establishing session keys),
- deriving dialogue keys for message protection (both confidentiality and integrity), and
- protecting systems security data such as PACs (Privilege Attribute Certificates).

The security mechanisms defined here allow a choice of algorithms which can be used for the different functions, depending on

- the needs of the functions, and
- the requirements for international deployment in countries which constrain how cryptography can be used and exported from countries where use of cryptography is controlled.

In some cases, export controls may require international versions of products to use shorter key lengths; therefore, a large number of combinations of algorithms and key lengths may be possible. For interoperability, both client and target must support the same algorithms and key lengths for these functions.

This specification defines a number of *cryptographic profiles*, where each profile identifies a set of algorithms with specified key lengths used by a mechanism for specified functions.

For example, the CSI-ECMA protocol defines a **NoDataConfidentiality** cryptographic profile which can use DES and RSA for protecting the security mechanism, but does not encrypt the ORB request/reply. (The profile for full security would use DES/64 for data confidentiality.)

Cryptographic profiles are identified by a value, represented in IORs as an unsigned short:

typedef unsigned short CryptographicProfile;

15.10.4.1 Key Establishment Algorithms

The algorithms used to establish the cryptographic session keys during security associations depend on the type of mechanism.

- Where the secret key (Kerberos-based) mechanism is used, either via the GSS Kerberos or CSI-ECMA protocol, the DES algorithm is used.
- When a public key mechanism is used, either via SPKM or CSI-ECMA protocol, the RSA algorithm is used.

15.10.4.2 Common Message Protection Algorithms

Even if different mechanisms and algorithms are used for key establishment, the same algorithms can be used for message protection.

- All CSI mechanisms have cryptographic profiles which include an MD5 hash of the data for integrity, though the hash, in some profiles may be signed or encrypted.
- All CSI mechanisms can use DES in CBC mode for message confidentiality.

15.10.4.3 Cryptographic Profiles Supported by CSI Protocols

A number of cryptographic profiles are defined for each CSI protocol. Further cryptographic profiles using different algorithms can be used with these protocols, but these are not part of this interoperability standard. A target may support several cryptographic profiles for a particular mechanism.

In all cases, support of a CSI protocol requires support for a cryptographic profile which provides integrity of user data, but not confidentiality, as such a profile is easier to deploy internationally. For example, the GSS Kerberos protocol always supports its *MD5* cryptographic profile. Other profiles may also be supported.

15.10.5 Security Name

The form of the security name depends on the security mechanism used. For example, it can be a Kerberos name or a Directory style name. Directory names conform to the string representation defined in [4].

The security name may be at the component level of the IOR or higher if shared between mechanisms. If there is a security mechanism tag, but no security name is present in the IOR, the IOR is improperly formatted and a CORBA::INV_OBJREF exception shall be raised when the IOR is used to specify the target of an operation.

15.10.6 Security Administration Domains

As defined in “Domains” on page 15-38, a security policy domain is a set of objects to which a security policy applies for a set of security-related activities and is administered by a security authority.

Security mechanisms are concerned with the security domains where users and other principals are administered, often by on-line authorities such as Authentication and Privilege Attribute Services. Often, this domain will be the enclosing domain encompassing secure invocation, access control, and other policy domains.

Note that some authorities may be off-line. For example, the Certification Authority used to issue certificates is often off-line.

The security mechanisms specified in this specification allow requests to cross domain boundaries. At the boundary, trust between the domains needs to be established. (The way this is done depends on the mechanism used.) Also, the scope of privileges may not always cross the domain boundary. This specification does not define how privileges are mapped on crossing domain boundaries, as this does not affect the protocol.

While all security mechanisms here include the concept of such domains, in Kerberos (used here as the secret key mechanism) these are known as realms. In this specification, the term realm is used in tokens using this mechanism.

15.10.7 Mapping of Common Elements to the SECIOP Protocol

The SECIOP protocol includes the tokens for context establishment and management and per-message tokens.

The context establishment tokens contain:

- Information associated with a principal, including at least an identity. (At CSI Level 2, there may be a range of privileges and a separate audit identity, if required.)
- Associated delegation information. Only simple delegation is mandatory to conform to this specification.
- Security information used to establish the client-target object security association.
- Security information used to establish the keys for message protection.

15.10.7.1 Basic Token Format

SECIOP messages include context and message protection tokens.

All CSI mechanisms are usable inside and outside the object environment. In line with standard practice outside the object environment, tokens are defined in ASN.1. and encoded for transmission using BER (in some cases, constrained to the DER subset of these). The token appears as a **sequence<octet>** in CDR-encoded SECIOP messages.

These tokens are enclosed within framing as follows:

```
[APPLICATION 0] IMPLICIT SEQUENCE {
    thisMech   MechType
                    -- MechType is OBJECT IDENTIFIER
    innerContextToken ANY DEFINED BY thisMech
                    -- contents mechanism-specific;
    }
```

Note – For conformance to GSS-API, only the initial context token has to use this token framing; however, in the CSI protocols, it applies to all tokens.The initial context token should include a mechanism version, as well as type. For CSI mechanisms, version numbers are in the mechanism-specific information such as the Kerberos ticket or CSI-ECMA PAC.

15.10.7.2 Inner Context Tokens

The same token types are used in the different CSI protocols, though not all protocols support all token types. The token types are defined below showing the relationship with GSS-API calls, as all CSI protocols can be implemented using GSS-API.

The inner context tokens used for security association establishment are:

InitialContextToken

This is sent by the initiator to a target, to start establishment of a security association in an SECIOP **EstablishContext** message. The token id is **01 00 (hex)**.
If GSS-API is being used, it is the value returned by the **GSS_Init_sec_context** call.

TargetResultToken

This is sent to the initiator by the target to complete establishment of the context in an SECIOP **CompleteEstablishContext** message. The token id is **02 00 (hex)**. It is returned by **GSS_Accept_sec_context**.

ContinueEstablishToken

This is sent either by the initiator or the target to continue context establishment in an SECIOP **ContinueEstablishContext** message. The token id is **03 00 (hex)** (in SPKM). It is returned by either the **GSS_Init_sec_context** call or the **GSS_Accept_sec_context** call.

ErrorToken

This is sent on detection of an error during security association establishment in an SECIOP **CompleteEstablishContext** or **ContinueEstablishContext** message. The token id is **03 00 (hex)** (except in SPKM where it is **04 00 (hex)**). It is returned by either the **GSS_Init_sec_context** call or the **GSS_Accept_sec_context** call.

The inner context token for message protection is the **message_protection_token** in the SECIOP **MessageInContext** message. This can take one of the following forms:

MICToken

This is sent either by the initiator or the target to verify the integrity of the user data sent in the following GIOP message (or message fragment). The token id is **01 01 (hex)** It is returned by **GSS_GetMIC**.

WrapToken

This is sent either by the initiator or the target. Encapsulates the input user data (optionally encrypted) along with integrity check values.The token id is **02 01 (hex)**. It is returned by **GSS_Wrap**.

This specification always uses **MIC** tokens for integrity and **Wrap** tokens for confidentiality. This may ease national use and export problems where only **MIC** tokens are supported.

The inner context token in the **DiscardContext** SECIOP message may optionally contain a **ContextDeleteToken**.

ContextDeleteToken

This is sent either by the initiator, or the target in an SECIOP **DiscardContext** message to release a Security Association. It is returned by **GSS_Delete_sec_context**.

15.10.8 CSI Protocols

This specification includes three protocols for different circumstances, as described in Section 15.8.6, "Key Distribution Types," on page 15-184.

In all cases, the appropriate section specifies the cryptographic profiles supported, and the contents of the SECIOP security tokens.

In all cases, the protocol as supported by OMG is a subset of the protocol defined in the source document. For example, in all protocols, channel bindings as defined in GSS-API (and specified in the underlying protocols) are not supported. This is because IP addresses cannot be trusted in current implementations; IP addresses are spoofable. Including the channel binding information would lead to a false sense of security about the source of the transmission.

The protocols described in this specification include SPKM, GSS Kerberos, and CSI-ECMA.

15.10.8.1 SPKM Protocol

The SPKM protocol supports CSI level 0. This is a public key-based protocol. The only client information transmitted is its security name. See Section 15.11, "SPKM Protocol," on page 15-231.

15.10.8.2 GSS Kerberos Protocol

The GSS Kerberos protocol supports CSI level 1. This is a secret key-based protocol. The only client information transmitted is its security name. See Section 15.12, “GSS Kerberos Protocol,” on page 15-234.

15.10.8.3 CSI-ECMA Protocol

The CSI-ECMA protocol also supports the privilege handling, separate **Auditid**, and delegation controls of CSI level 2. Subschemes within this protocol support the three key distribution options: secret, public, and hybrid. See Section 15.13, “CSI-ECMA Protocol,” on page 15-237 for additional information.

To support this flexibility, the **initial_context_token** is split into three parts; therefore, the attributes for access control are independent of the key distribution method, and this is independent of the cryptography used for message protection. The token contains:

- Authorization information - attributes of a principal are held in a Privilege Attribute Certificate (PAC) with any associated information needed for delegation and other controls. This is independent of the way the communications are protected; therefore, it is usable with different key distribution methods.
- Security information needed to establish the association. The form that the security information takes depends on the key distribution method used. It is a Kerberos ticket if this is secret key-based; it is a profile of the **SPKM_REQ** token for public key mechanisms. In both cases, there is a link between this and the PAC. Changing the security mechanism mainly just requires replacing this part of the token.
- Dialogue key packages to establish confidentiality and integrity keys.

15.11 SPKM Protocol

This section specifies the SPKM protocol, a simple public-key GSS-API mechanism. It is based on SPKM as defined in [20]. SPKM protocol provides CSI level 0 functionality only and the purpose is to allow the adoption of a simple security infrastructure without undue complexity or overhead.

SPKM has two separate GSS-API mechanisms, **SPKM_1** and **SPKM_2**, whose primary difference is that **SPKM_2** requires the presence of secure timestamps for the purpose of replay detection during context establishment and **SPKM_1** does not. **SPKM_1** is the mandatory mechanism for conformance to the SPKM protocol while **SPKM_2** is the optional mechanism.

Specifically, it defines the required information for encoding a secure interoperability IOR and defines the token formats used by the SECIOP protocol.

15.11.1 Cryptographic Profiles

The following cryptographic profiles are supported with this mechanism:

15.11.1.1 MD5_RSA

Specifies use of the SPKM mechanism to provide data integrity and authenticity by computing an RSA signature on the MD5 hash of that data. The default SPKM key establishment algorithm is used (i.e., the context key is generated by the initiator, encrypted with the RSA public key of the target, and sent to the target). Note that **MD5_RSA** is a mandatory integrity and authenticity algorithm for SPKM.

15.11.1.2 MD5_DES_CBC

Specifies use of the SPKM mechanism to provide data integrity by encrypting, using DES in CBC mode, the MD5 hash of that data. The default SPKM key establishment algorithm is used.

15.11.1.3 DES_CBC

Specifies use of the SPKM mechanism to provide data confidentiality by using DES in CBC mode. The default key establishment algorithm is used.

15.11.1.4 MD5_DES_CBC_SOURCE

Specifies use of the SPKM mechanism to provide data integrity by encrypting, using DES in CBC mode, the MD5 hash of that data. The default key establishment algorithm is used plus source authentication information is also encrypted with the target's public key.

15.11.1.5 DES_CBC_SOURCE

Specifies use of SPKM mechanism to provide data confidentiality by using DES in CBC mode. The default key establishment algorithm is used plus source authentication information is also encrypted with the target's public key.

Values for these cryptographic profiles are assigned in Appendix Section A.10, “General Security Data Module,” on page 15-283.

15.11.2 IOR Encoding

The security tags in the IOR are encoded. The component data member associated with the **SPKM_1** and **SPKM_2** mechanism tags is a **struct**, defined as follows:

```
struct <mechanism_name> {
    AssociationOptions                    target_supports;
    AssociationOptions                    target_requires;
    sequence <CryptographicProfile> crypto_profiles;
    sequence<octet>                       security_name;
};
```

mechanism_name can be either **SPKM_1** or **SPKM_2** and **security_name** must contain a valid **X.500** distinguished name represented as a string conforming to [4]. For example, it could be **“cn=Andrew Rust, ou=Home Office, o=Acme Widgets Inc., c=CA"**.

All tag components must be encapsulated using CDR encoding.

15.11.3 Using SPKM for SECIOP

When the SPKM protocol is chosen as the security mechanism for invoking an object, the SECIOP protocol carries the information described in this section. This protocol is a profile of the SPKM GSS-API mechanism as defined in [20].

All SPKM tokens are encoded according to the general format described in Section 15.10.7, “Mapping of Common Elements to the SECIOP Protocol,” on page 15-228.

The **innerContextTokens** are described in the following sections. All **innerContextTokens** are encoded using ASN.1 BER (constrained, in the interests of parsing simplicity, to the DER subset defined in [22]).

The SPKM GSS-API mechanism is identified by an OBJECT IDENTIFIER representing "**SPKM_1**" or “**SPKM_2**.” **SPKM_1** uses random numbers for replay detection during context establishment and **SPKM_2** uses timestamps (note that for both mechanisms, sequence numbers are used to provide replay and out-of-sequence detection during the context, if this has been requested by the application). **SPKM_1** OBJECT IDENTIFIER is **1.3.6.1.5.5.1.1** and **SPKM_2** OBJECT IDENTIFIER is **1.3.6.1.5.5.1.2**.

15.11.3.1 The Initial Context Token

The **initial_context_token** carried within an **EstablishContext** SECIOP message is encoded according to the general framework and confirms to the **SPKM-REQ** token as described in [20] Section 3.1.1.

In the **initial_context_token**, channel bindings are required to be ZERO (**GSS_C_NO_BINDINGS**).

The **GSS_C_DELEG_FLAG** is required to be FALSE (no delegation is supported).

The **GSS_C_MUTUAL_FLAG** is TRUE if it requires both parties to authenticate itself and FALSE (the default) if only one party is required to authenticate itself.

15.11.3.2 The Final Context Token

The **final_context_token** carried within a **CompleteEstablishContext** SECIOP message is encoded according to the **SPKM-REP-TI** token as defined in [20] Section 3.1.2 or the **SPKM-ERROR** token as defined in [20] Section 3.1.4.

15.11.3.3 The Continuation Context Token

The **continuation_context_token** carried within a **ContinueEstablishContext** SECIOP message is encoded according to the **SPKM-REP-TI** token or the **SPKM-REP-IT** token as defined in [20] Section 3.1.3 or the **SPKM-ERROR** token.

15.11.3.4 The Message Protection Token

The **message_protection_token** carried within an SECIOP **MessageInContext** message is encoded according to the **SPKM-MIC** token (for integrity) or **SPKM-WRAP** token (for confidentiality) as defined in [20] Section 3.2.

15.11.3.5 The Context Delete Token

The **context_delete_token** carried within an SECIOP **DiscardContext** message is encoded according to the **SPKM-DEL** token as defined in [20] Section 3.2.3.

15.12 GSS Kerberos Protocol

This section specifies the GSS Kerberos protocol. It is based on the GSS Kerberos specification [12] which itself is based on Kerberos V5 as defined in [13]. This specification refers to, rather than repeats, information in [12] and [13].

This section defines the required information for encoding the mechanism-specific information in the IOR and the token formats used by the SECIOP protocol.

15.12.1 Cryptographic Profiles

The following cryptographic profiles are supported with this mechanism:

15.12.1.1 DES_CBC_DES_MAC

Specifies use of the Kerberos V5 mechanism with DES MAC message digest for integrity and DES in CBC mode for confidentiality.

15.12.1.2 DES_CBC_MD5

Specifies use of the Kerberos V5 mechanism with MD5 message digest for integrity and DES in CBC mode for confidentiality.

15.12.1.3 DES_MAC

Specifies use of the Kerberos V5 mechanism with DES MAC message digest for integrity.

15.12.1.4 MD5

Specifies use of the Kerberos V5 mechanism with a DES-encrypted MD5 message digest for integrity.

Values for these cryptographic profiles are assigned in Appendix Section A.10, “General Security Data Module,” on page 15-283.

15.12.2 Mandatory and Optional Cryptographic Profiles

ORB implementations claiming conformance to the GSS Kerberos protocol must implement at least the MD5 profile. Conformant ORBs may, but are not required to, implement the remaining cryptographic profiles defined in this specification.

15.12.3 IOR Encoding

The security tags in the IOR are encoded. Both security name and association options tags may appear in the IOR and be shared between mechanisms.

The component data member associated with the **KerberosV5** mechanism tag is a **struct** defined as follows:

```
struct KerberosV5 {
    AssociationOptions                  target_supports;
    AssociationOptions                  target_requires;
    sequence<CryptographicProfile>  crypto_profiles;
    sequence<octet>                     security_name;
};
```

security_name shall contain a valid Kerberos Principal Name of type **GSS_KRBV5_NT_PRINCIPAL_NAME**, which is defined in [12].

All tag components must be encapsulated using CDR encoding.

15.12.4 SECIOP Tokens

When the GSS-Kerberos protocol is chosen as the security mechanism for invoking an object, the SECIOP protocol carries the information described in this section. All Kerberos tokens are encoded according to the general format.

The OBJECT IDENTIFIER for Kerberos V5 is **1.3.5.1.2** until [12] is advanced to a Proposed Standard RFC when it will be changed to **1.2.840.113554.1.2.2**.

Each individual token is distinguished by the data carried in the ANY field of this general framework.

15.12.4.1 The Initial Context Token

The **initial_context_token** carried within an **EstablishContext** SECIOP message is encoded according to the general framework and conforms to the unencrypted authenticator message as described in [12] Section 1.1.1.

Note – Channel bindings are required to be ZERO (**GSS_C_NO_BINDINGS**) in this specification (see Section 15.10.8, "CSI Protocols," on page 15-230).

The **GSS_C_DELEG_FLAG** is set when either the client has called **set_security_features** specifying **SecDelModeSimpleDelegation** or when an administrator has called **set_delegation_mode** with a value of **SecDelModeSimpleDelegation** on a domain to which the target object belongs. The optional "Deleg" field, if present, includes a forwardable Ticket Granting Ticket (TGT) representing the delegated credentials of the client sending the **EstablishContext** message.

The **GSS_C_MUTUAL_FLAG** is set when either the client has called **set_association_options** specifying a value of **EstablishTrustInTarget** or an administrator has called **set_association_options** with a value of **EstablishTrustInTarget** on the domain to which the target belongs.

The **GSS_C_REPLAY_FLAG** and **GSS_C_SEQUENCE_FLAG** are generally clear as they can cause incorrect replay and misordering detection in a multi-threaded environment (see Section 15.10.3, "Association Options," on page 15-225).

Note – The current GSS Kerberos implementation available without cost from MIT does not support replay detection.

15.12.4.2 The Final Context Token

The **final_context_token** carried within a **CompleteEstablishContext** SECIOP message is encoded according to the formats defined in [12] Section 1.1.2.

15.12.4.3 The Continuation Context Token

Kerberos V5 does not use the **ContinueEstablishContext** message and therefore does not define the **continuation_context_token** format. If the Kerberos V5 mechanism is amended in the future to support mechanism negotiation, support of the **ContinueEstablishContext** message would be necessary and thus definition of the **continuation_context_token** would be required.

15.12.4.4 The Message Protection Token

The **message_protection_token** carried within an SECIOP **MessageInContext** message is encoded according to the formats defined in [12] section 1.2.

15.13 CSI-ECMA Protocol

This section defines the CSI-ECMA protocol. It is based on the ECMA GSS-API mechanism as defined in ECMA-235, though it is a significant subset of that. It supports all CSI levels (0, 1, and 2), and provides three options for key distribution:

1. A secret key option using Kerberos data structures.
2. A hybrid option where secret keys are used within an administrative domain, but public keys are used between domains.
3. A public key option which uses public key technology for key distribution both within and between domains.

This section includes the full definition of the CSI-ECMA protocol so that it can be read without reference to ECMA 235. The CSI-ECMA protocol is a subset of ECMA 5. It is very similar to the SESAME profile as described in [16].

The CSI-ECMA protocol supports the CORBA Security Level 2 facilities. It is designed to be extensible as new facilities (for example, new delegation options) are agreed upon in the future, and further key distribution options. It is also designed to respond to the requirements of international deployment such as minimal confidentiality (only keying information needs to be encrypted), use of anonymous audit (a separate **audit_id** can be transmitted), and choice of cryptography for message protection (including strong integrity, weak confidentiality).

The structure of the initial context token is key to providing this flexibility. It is separated into three parts:

1. Authorization information.
2. Information concerned with establishing the security association using one of the supported key distribution options.
3. Information concerned with generating the dialogue keys for message protection.

15.13.1 Concepts

15.13.1.1 Separation of Concerns

The initial context token transmitted in the SECIOP EstablishContext message on setting up a security association contains a number of parts with limited links between them. This is so that the different parts can be varied independently of each other. The three main parts are:

1. Authorization information - the Privilege Attribute Certificate (PAC) which contains the privileges used for access control and other attributes such as the audit id. Associated with this are delegation and other controls. Therefore, this is concerned with the access control and delegation policies, but is mainly independent of the key establishment and message protection mechanisms. The PAC can be updated to affect these policies independently of mechanisms. (The size of the PAC may be

significant; therefore, it is not confidentiality-protected, as this may cause regulatory problems.) Privilege and other attributes in PACs are described in Section 15.13.2, “Security Attributes,” on page 15-238.

2. Target key block - used to provide the information needed to establish the security association between client and target. Secret key or public key technology (or some hybrid of these) may be used. The result is always a “basic” key from which dialogue keys to protect application messages can be derived. Therefore, this is concerned with the mechanism for establishing trust and distributing keys. This can be varied independently of the authorization policies and the message protection methods. Key establishment methods are described in Section 15.13.5, “Key Distribution Schemes,” on page 15-239.

3. Dialogue key packages which control how dialogue keys to protect messages are derived from the basic key. Note that this is largely independent of the key distribution method (i.e., public key technology may be used to establish secret keys for dialogue protection).

15.13.2 Security Attributes

15.13.2.1 Privilege Attributes

The CSI-ECMA protocol allows a range of privilege attributes in a Privilege Attribute Certificate (PAC) transmitted between the client and target object. These privileges then can be used for access control.

Privilege attributes which can be carried in the PAC at level 2 are defined in Appendix Section A.11.1, “Attribute Types,” on page 15-307 and include all those defined in the CORBA Security specification.

A vendor or user enterprise may also define its own privilege attributes (if the particular implementation allows this) and use them for access control.

In line with the CORBA Security specification, each privilege attribute has a defining authority which identifies the authority responsible for defining the semantics of the value of the security attribute. This can be included for each privilege attribute in the PAC and in this case, there could be a different defining authority for each privilege.

It is often the case that most attributes in the PAC come under the same defining authority which is the authority that issued the PAC. If the PAC, as transmitted, does not have defining authorities for some attributes, then the issuing authority of the PAC is considered to be the defining authority.

15.13.2.2 Miscellaneous Attributes

This specification allows other types of security attributes to be carried in the PAC under the general heading of miscellaneous attributes. In CSI-ECMA, the only type of miscellaneous attribute supported is the audit identity.

15.13.3 Target Access Enforcement Function

The security processing functionality at the target is split between the target application and the target access-enforcement function (**targetAEF**). ISO (ISO/IEC 10181-3) defines an access enforcement function collocated with the target application which controls access to a target application. This has a number of advantages including:

- The security critical code is isolated which makes security evaluation simpler.
- Long term keys can be shared between applications/objects. This can simplify administration (as there are less keys) and allow re-use of keying information when accessing another application/object sharing this **targetAEF**.

The **targetAEF** is responsible for setting up the security association, including validating the PAC and releasing the keys for message protection.

15.13.4 Basic and Dialogue Keys

The exchanges between client and target are secured using a two-level key scheme in which a distinction is made between basic and dialogue keys.

A basic key is a temporary key established between a client and the target (actually, the **targetAEF**). The basic key is used for integrity protection of the PAC and associated information, its own key establishment information, and the information used to establish the dialogue keys. The basic key is established by the client sending information to the target in the **targetKeyBlock**. This can take different forms, depending on the key distribution method used.

A dialogue key is a temporary key established between the client and target and is used to protect the requests and responses. Separate dialogue keys can be established for integrity and confidentiality protection, enabling different strengths of mechanism to be configured. The information required to derive the dialogue keys is transmitted in the **Dialogue** key package. Typically, dialogue keys are constructed from the basic key using a one way algorithm.

15.13.5 Key Distribution Schemes

The CSI-ECMA protocol allows a choice of key distribution methods for establishing a client-target security association including the basic key. The content of the **targetKeyBlock** depends on the scheme used.

The key distribution schemes depend on the existence of long term cryptographic keys. Both secret (symmetric) and public (asymmetric) key technology can be used. When secret keys are used, a key is shared between the target and its Key Distribution Service (KDS). When public keys are used, the private key is kept by the principal and the public key held in a certificate, in a directory or elsewhere.

Initiators may also possess symmetric or asymmetric keys established as the result of an earlier authentication.

This CSI-ECMA specification defines three key distribution schemes. These are described below and are identified by a name and an architectural option number. Other schemes are possible as extensions to this as described in ECMA-235.

15.13.5.1 Basic Symmetric Key Distribution Scheme

In this scheme, the client and target each share different secret keys with the same Key Distribution Server. The scheme name for this is: **symmIntradomain**. The architectural option number is **2**.

To establish the association between the client and target, the client obtains a **targetKeyBlock** from its KDS containing a basic key encrypted under the target's long term key. On receipt of the targetKeyBlock, the target can extract the basic key from it.

In this case, the targetKeyBlock is a Kerberos ticket.

15.13.5.2 Symmetric Key Distribution with Asymmetric KDS

In this scheme, the initiator shares a secret key with its KDS and the target shares a secret key with its KDS (which is different). In addition, each KDS possesses a private/public key pair. The scheme name for this is: **hybridInterdomain**. The architectural option number is **3**.

To establish the client-target association, the client gets a targetKeyBlock from its KDS containing the basic key encrypted under a temporary key and the temporary key encrypted under the target's KDS public key. The **targetKeyBlock** is also signed using the initiator's KDS private key.

On receipt of the **targetKeyBlock**, the target transmits it to its KDS and gets back the basic key encrypted under the long term secret key it shares with its KDS.

15.13.5.3 Full Public Key Scheme

In this scheme, both client and target possess private/public keys. Neither use a KDS. The scheme name for this is: **asymmetric**. The architectural option number is **6**.

To establish the client-target association, the client constructs a **targetKeyBlock** containing a basic key encrypted under the target's public key. The target key block is signed with the client's private key. On receipt of the **targetKeyBlock**, the target directly establishes a basic key from it.

15.13.6 Cryptographic Algorithms and Profiles

Cryptographic and hashing algorithms are used for various purposes. This section categorizes the algorithms according to usage so that client and targets can determine more easily if they have the cryptographic support required to allow interoperation. The categorization then is refined into cryptographic profiles that can be incorporated into specific mechanism identifiers. The mechanism identifiers with cryptographic profiles then can be carried in the IOR. Table 15-15 summarizes the different uses to which algorithms are put.

Table 15-15 Summary of Algorithm Usage

Use Reference	Description of Use	Type of Algorithm
2	PAC protection using signature	OWF + asymmetric signature
3	basic key usage	confidentiality and integrity
4	integrity dialogue key derivation	OWF
5	integrity dialogue key usage	symmetric integrity
6	CA public keys	OWF + asymmetric signature
7	encryption using shared long term symmetric key	symmetric confidentiality
8	name hash to prevent ciphertext stealing	OWF
9	asymmetric basic key distribution	asymmetric encryption
10	key establishment within **SPKM_REQ**	(fixed value)
11	confidentiality dialogue key derivation	OWF
12	confidentiality dialogue key use	symmetric confidentiality

The algorithms can now be further categorized into broader classes, as shown in the following table.

Table 15-16 Summary of Algorithm Classes

Class 1:	symmetric for security of mechanism:	uses 3, 5, 7
Class 2:	all OWFs:	uses 2, 4, 6, 8, 11
Class 3:	internal mechanism asymmetric, encrypting:	use 9
Class 4:	internal mechanism asymmetric, non encrypting:	use 2
Class 5:	CA's asymmetric non-encrypting:	use 6
Class 6:	data confidentiality, symmetric:	use 12

Use 10 is a fixed value and does not contribute to mechanism use options.

Based on these classes, the following cryptographic algorithm usage profiles are defined. Other profiles are possible and can be defined as required. Note that symmetric algorithm key sizes are included in this profiling, thus **DES/64** indicates **DES** with a 64-bit key.

Table 15-17 Cryptographic Algorithm Usage Profiles

Class	**Profile 1 Full**	**Profile 2 no data confidentiality**	**Profile 3 low-grade confidentiality**	**Profile 5 defaulted**
Class 1	DES/64	DES/64	RC4/128	separately agreed default
Class 2	MD5	MD5	MD5	separately agreed default
Class 3	RSA	RSA	RSA	separately agreed default
Classes 4 and 5	RSA	RSA	RSA	separately agreed default
Class 6	DES/64	None	RC4/40	separately agreed default

Table key:

- Profile 1 provides full security, using standard cryptographic algorithms with common accepted key sizes.
- Profile 2 is the same, but without supporting any confidentiality of user data.
- Profile 3 provides low-grade confidentiality. In some countries, products using this are exportable without restriction; in others, they are more easily exportable/importable.
- Profile 5 uses algorithms identified by a separately specified default. It is intended for use by organizations who wish to use their own proprietary or government algorithms by separate agreement or negotiation.

15.13.7 PAC Protection and Delegation - Outline

The ECMA protocol provides a number of ways to protect a principal's credentials, as held in a PAC. In CSI-ECMA, a digital signature is used, as this allows a target system to check what Security Authority authorized use of these privileges, without relying on the transitive trust needed for sealed PACs crossing domain boundaries. Encrypted PACs are not included in this profile.

There may also be controls on where the PAC may be delegated and used.

Protection method fields in the PAC specify where this PAC can be used and whether it can be used by the specified targets only (for example, allowing use of the privileges for access control) or whether that target can also delegate it.

Protection method fields are grouped together into method groups. The protection method check is passed if all the method fields in any one of the method groups is passed.

15.13.8 PPID Method

This method protects the **PAC** from being stolen, by restricting the initiators who can use the **PAC**.

When no other method group is present, it permits the **PAC** to be used only by the client entity to which it was originally issued (i.e., it prevents delegation). However, a **PAC** with a **PPID** will be delegatable if delegation is permitted by a **PV/CV** method.

A **PPID** identifying the initiating principal is put in the **PAC** by the Privilege Attribute (or other security) Service, according to policy or client request. The same/related information is also supplied as part of the **targetKeyBlock** so that the target can check that the entity which sent this token is the same entity which is entitled to use the PAC.

The **PPID** is a security attribute whose value in the CSI-ECMA protocol can take one of two forms, depending on the key distribution scheme used by the initiator.

- When the initiator has a secret key, the **PPID** is a random bit string which is also sent in the authorization field of the Kerberos ticket. This ticket is sent as part of the targetKeyBlock and can be checked to come from this client.
- For the public key scheme, the **PPID** contains the certificate serial number and **CA** name for the initiator's **X.509** public key certificate. The **targetKeyBlock** sent to the target is signed using this initiator's private key.

15.13.9 PV/CV Delegation Method

This method prevents the **PAC** from being stolen and at the same time controls whether (and where) it can be delegated. The method field in the **PAC** contains a protection value (**PV**) which is a one-way function of a Control Value (**CV**).

A **PAC** will be accepted by the target (subject to other controls in the **PV**'s method group) if the client proves knowledge of the **CV** by passing it (encrypted) as part of the initial context token. A method group contains at most one **PV** value.

In the simplest case, the method group contains just the **PV** and the target can delegate the **PAC** if it receives the **CV**.

The **PV/CV** method can be used for more selective targeting of the **PAC** also. A method group can include qualifier attributes which specify where the **PAC** can be used. **Qualifier** attributes can specify which principals can receive the **PAC** as a target and which can act as both delegate and target. These principals can be specified by their identities (though the protocol is extensible for other options such as a group/domain to which they belong).

For the simpler case, delegation can be prevented by setting the delegation mode to **Security::SecDelModeNoDelegation**. This will cause the client to send the **PAC** without the **CV**.

Note – The protocol allows more than one method group in the **PAC**, each with its own **PV/CV**. This can be used by a client or intermediate object in a chain to further restrict who can use the **PAC**, by failing to send some of the **CV**s. However, this specification does not include any operations for restricting delegation in this way, so it is not possible to exploit this capability.

15.13.9.1 Restrictions

Other restrictions may be included in the PAC. An ORB conforming to this specification does not have to generate these restrictions, but will reject PACs with mandatory restrictions which it does not understand or cannot process.

15.13.10 Mechanism Identifiers and IOR Encoding

All tag component data in the IOR must be encapsulated using CDR encoding.

Mechanism identifiers for the CSI-ECMA protocol have up to three parts, as follows:

1. ***Protocol identifier*** - this is CSI-ECMA.
2. ***Architectural option*** - this identifies the architectural option (i.e., the key distribution method used when establishing security associations). If absent, the default option is used.
3. ***Cryptographic profile*** - this identifies the cryptographic profile as defined above. If absent, a default is used.

In the IOR, the mechanism name in the struct of the **TAG_x_SEC_MECH** is:

CSI-ECMA_<architectural option>

where the architectural options supported are Secret, Hybrid, and Public; therefore, mechanism names are **CSI_ECMA_Secret**, **CSI_ECMA_Hybrid**, and **CSI_ECMA_Public**.

These values could also be negotiated using a generic mechanism negotiation scheme such as that in [19] in the future, but are in the IOR for the current CSI specification.

15.13.11 Security Names

This protocol uses two forms of security names:

1. Directory names (DNs) are used where public key technology is used, as this is the form of name used in X509 certificates.
2. Kerberos names are used where secret key technology is used, as this is the form of name used by Kerberos.

15.13.11.1 Kerberos Naming

An entity that uses the normal Kerberos V5 authentication is given a printable Kerberos principal name of the form:

<principal_name>@realm_name>

Note – Components of a name can be separated by "/". The separator @ signifies that the remainder of the string following the @ is to be interpreted as a realm identifier. If no @ is encountered, the name is interpreted in the context of the local realm. Once an @ is encountered, a non-null realm name, with no embedded "/" separators must follow. The "/" character is used to quote the character that follows immediately.

15.13.11.2 Directory Naming

Where public key technology supported by Directory Certificates is used, entities are given DNs. Such names are normally transmitted as directoryNames. At interfaces, they are strings built from components separated by a semicolon. The standardized keywords supported are:

CN (common-name)
S (surname)
OU (organization unit)
O (organization)
C (country)

An example of a supported DN is:

CN=Martin;OU=Sesame;O=Bull;C=fr

There is no general rule for mapping the Directory name of an entity onto its Kerberos principal name. An explicit mapping is provided in a principal's Directory Certificate using the extensions field of the extended Directory Certificate syntax (version 3) to carry the principal's Kerberos name.

The syntax of the login name is imported from the Kerberos V5 GSS-API mechanism. The form of name is referred to using the symbolic name: **GSS_KRB5_NT_PRINCIPAL**. Syntax details are given in [12].

15.13.12 SECIOP Tokens When Using CSI-ECMA

All SECIOP security tokens conform to the basic token format defined in "Basic Token Format" on page 15-228. The object identifier for the **MechType** is of the form:

{generic_CSI_ECMA_mech (y) (z)}

where the value for **generic_CSI_ECMA_mech** is **1.3.12.0.235.4** and the values of y and z, if present, represent the architectural option number and cryptographic profile numbers. Both **y** and **z** can be defaulted.

The **innerContextToken** of the SECIOP message may be any of the tokens defined in "Inner Context Tokens" on page 15-229.

For context establishment, tokens are:

InitialContextToken - Sent by the initiator to a target, to start the process of establishing a Security Association.

TargetResultToken - Sent to the initiator by the target, if needed, following receipt of an Initial Context Token.

ErrorToken - Sent by the target on detection of an error during Security Association establishment.

The per-message tokens are:

MICToken - Sent either by the initiator or the target to verify the integrity of the user data sent separately.

WrapToken - Sent either by the initiator or the target. Encapsulates the input user data (optionally encrypted) along with integrity check values.

A **ContextDeleteToken** may also be used either by the initiator or the target to release a Security Association.

This definition uses ASN.1 types from other standards (e.g., the ISO definition of a Certificate). These types are detailed in Annex E of ECMA-235.

15.13.13 Initial Context Token

The initial context token contains:

- General information such as the token id, **contextFlags** (**delegation, replay-detect** etc.), **utcTime**, **seq-number**, etc.
- A **targetAEF** part to be passed to the target access enforcement function. This includes the PAC and associated CVs, target key block, and dialogue key package.
- A seal.

<table>
<tr><td rowspan="2">token id. etc.</td><td colspan="3">target AEF part
(used by target to enforce policy)</td><td rowspan="2">seal</td></tr>
<tr><td>pac & CVs
(initiating and/or delegate principal's authorization and delegation information)</td><td>target Key Block
(information needed to establish the association)</td><td>dialogue Key Block
(information used to establish message protection key - integrity and confidentiality)</td></tr>
</table>

Figure 15-61 Initial Context Token

```
InitialContextToken ::=  SEQUENCE{
        ictContents        [0]   ICTContents,
        ictSeal            [1]   Seal
}
```

ictContents - Body of the initial context token.

ictSeal - Seal of ictContents computed with the integrity dialogue key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by **integDKUseInfo** in the **dialogueKeyBlock** field of the initial context token.

```
ICTContents ::= SEQUENCE {
        tokenId                 [0]   INTEGER, -- shall contain X'0100'
        SAId                    [1]   OCTET STRING,
        targetAEFPart           [2]   TargetAEFPart,
        targetAEFPartSeal       [3]   Seal,
        contextFlags            [4]   BIT STRING {
                                          delegation (0),
                                          mutual-auth (1),
                                          replay-detect (2),
                                          sequence (3),
                                          conf-avail (4),
                                          integ-avail (5)
                                      }
        utcTime                 [5]    UTCTime              OPTIONAL,
        usec                    [6]    INTEGER              OPTIONAL,
        seq-number              [7]    INTEGER              OPTIONAL,
        initiatorAddress        [8]    HostAddress          OPTIONAL,
        targetAddress           [9]    HostAddress          OPTIONAL
}
```

tokenId - Identifies the initial-context token. Its value is **01 00 (hex)**

SAId - A random number for identifying the Security Association being formed; it is one which probably has not been used previously. This random number is generated by the initiator and processed by the target as follows:

- If no **targetResultToken** is expected, the SAId value is taken to be the identifier of the Security Association being established (if this is unacceptable to the target, then an error token with etContents value of **gss_ses_s_sg_sa_already_established** must be generated).
- If a **targetResultToken** is expected, the target generates its random number and concatenates it to the end on the initiator's random number. The concatenated value is then taken to be the identifier of the Security Association being established.

targetAEFPart - Part of the initial-context token to be passed to the target access enforcement function. This is defined below and includes PAC, basic, and dialogue key packages.

targetAEFPartSeal - Seal of the **targetAEFPart** computed with the basic key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by algorithm profile in the mechanism option.

contextFlags - Combination of flags that indicates context-level functions requested by the initiator.

Flag	Indicates that ...
delegation	when set to 0, the initiator explicitly forbids delegation of the PAC in the **targetAEFPart**.
mutual-auth	mutual authentication is requested.
replay-detect	replay detection features are requested to be applied to messages transferred on the established Security Association.
sequence	sequencing features are requested to be enforced to messages transferred on the established Security Association.
conf-avail	a confidentiality service is available on the initiator side for the established Security Association.
integ-avail	an integrity service is available on the initiator side for the established Security Association.

utcTime - The initiator's UTC time.

usec - Microsecond part of the initiator's time stamp. This field along with utcTime are used together to specify a reasonably accurate time stamp.

seq-number - When present, this specifies the initiator's initial sequence number; otherwise, the default value of **0** is to be used as an initial sequence number.

initiatorAddress - Initiator's network address part of the channel bindings. This field is present only when channel bindings are transmitted by the caller to the mechanism implementation. Conformant ORBs do not need to generate this field.

targetAddress - Target's network address part of the channel bindings. This field is present only when channel bindings are transmitted by the caller to the implementation.

15.13.13.1 TargetAEF Part

```
TargetAEFPart ::= SEQUENCE {
    pacAndCVs              [0]  SEQUENCE OF CertandECV OPTIONAL,
    targetKeyBlock         [1]  TargetKeyBlock,
    dialogueKeyBlock       [2]  DialogueKeyBlock,
    targetIdentity         [3]  Identifier,
    flags                  [4]  BIT STRING {
                                    delegation      (0)
                                }
}
```

pacAndCVs - The initiator **ACI** to be used for this Security Association. This field is not present when the association does not require any **ACI**. This field contains the **PAC** together with associated **PAC** protection information. When only simple delegation is supported, one of these should be present.

If composite delegation options are supported, this field will contain more than one **PAC**. For example, for the initiator plus immediate invoker case, the initiator's **PAC** would be present (with **CV**s) and the immediate invoker's (with a **PPID**).

targetKeyBlock - The **targetKeyBlock** carrying the basic key to be used for the Security Association being established.

dialogueKeyBlock - A dialogue key block used by the **targetAEF** along with the basic key to establish an integrity dialogue key and a confidentiality dialogue key for per-message protection over the Security Association being established.

targetIdentity - The identity of the intended target of the Security Association. Used by the **targetAEF** to validate the **PAC**. It can also be used by the **targetAEF** to help protect the delivery of dialogue keys.

flags - Flags required by the **targetAEF** for its validation process. **flags** contains only a delegation flag, the value of which is the same as the value of delegation flag in **contextFlag** field of **ictContents**. When the flag is set, all ECVs sent in **pacAndCVs** are made available to the target. Other bits are reserved for future use.

15.13.14 TargetResultToken

This token is returned by the target if the mutual-req flag is set in the Initial Context Token. It serves to authenticate the target to the initiator since only the genuine target could derive the integrity dialogue key needed to seal the **TargetResultToken**.

```
TargetResultToken ::=  SEQUENCE{
        trtContents             [0]    TRTContents,
        trtSeal                 [1]    Seal
}

TRTContents ::= SEQUENCE {
        tokenId                 [0]    INTEGER,    -- shall contain X'0200'
        SAId                    [1]    OCTET STRING,
        utcTime                 [5]    UTCTime         OPTIONAL,
        usec                    [6]    INTEGER         OPTIONAL,
        seq-number              [7]    INTEGER         OPTIONAL,
}
```

Note – There is no field for returning certification data here. This is because any such data that may be required is assumed to be returned at the conclusion of mechanism negotiation.

trtContents - This contains only administrative fields, identifying the token type, the context, and providing exchange integrity.

seq-number - When present this specifies the target's initial sequence number; otherwise, the default value of **0** is to be used as an initial sequence number.

The other administrative fields are as described previously.

trtSeal - Seal of **trtContents** computed with the integrity dialogue key. Only the **sealValue** field of the **Seal** data structure is present. The cryptographic algorithms that apply are specified by **integDKUseInfo** in the **dialogueKeyBlock** field of the initial context token.

15.13.15 ErrorToken

An error token may be returned, as follows:

```
ErrorToken ::=  {
        tokenType               [0]   OCTET STRING VALUE X'0400',
        etContents              [1]   ErrorArgument,
}
```

etContents - Contains the reason for the creation of the error token. The different reasons are given as minor status return values.

```
ErrorArgument ::= ENUMERATED {
        gss_ses_s_sg_server_sec_assoc_open                  (1),
        gss_ses_s_sg_incomp_cert_syntax                     (2),
        gss_ses_s_sg_bad_cert_attributes                    (3),
        gss_ses_s_sg_inval_time_for_attrib                  (4),
        gss_ses_s_sg_pac_restrictions_prob                  (5),
        gss_ses_s_sg_issuer_problem                         (6),
        gss_ses_s_sg_cert_time_too_early                    (7),
        gss_ses_s_sg_cert_time_expired                      (8),
        gss_ses_s_sg_invalid_cert_prot                      (9),
        gss_ses_s_sg_revoked_cer                            (10),
        gss_ses_s_sg_key_constr_not_supp                    (11),
        gss_ses_s_sg_init_kd_server_ unknown                (12).
        gss_ses_s_sg_init_unknown                           (13),
        gss_ses_s_sg_alg_problem_in_dialogue_key_block      (14),
        gss_ses_s_sg_no_basic_key_for_dialogue_key_block    (15),
        gss_ses_s_sg_key_distrib_prob                       (16),
        gss_ses_s_sg_invalid_user_cert_in_key_block         (17),
        gss_ses_s_sg_unspecified                            (18),
        gss_ses_s_g_unavail_qop                             (19),
        gss_ses_s_sg_invalid_token_format                   (20)
}
```

15.13.16 Per-Message Tokens

The syntax of the **message_protection_token** in SECIOP messages has the same general structure for both **MIC** and **Wrap** tokens:

```
PMToken ::=  SEQUENCE{
        pmtContents             [0]   PMTContents,
        pmtSeal                 [1]   Seal
                -- seal over the pmtContents being protected
}

PMTContents ::= SEQUENCE {
        tokenId                 [0]   INTEGER, -- shall contain X'0101'
        SAId                    [1]   OCTET STRING,
```

```
        seq-number          [2]   INTEGER        OPTIONAL
        userData            [3]   CHOICE {
                                        plaintext        BIT STRING,
                                        ciphertext       OCTET STRING  OPTIONAL
                                  }
        directionIndicator  [4]   BOOLEAN       OPTIONAL
}
```

pmtContents

tokenId
SAId - A random number for identifying the Security Association being formed; it is one which probability has not been used previously. This random number is generated by the initiator and processed by the target as follows:

- If no **targetResultToken** is expected, the **SAId** value is taken to be the identifier of the Security Association being established (if this is unacceptable to the target, then an error token with **etContents** value of **gss_ses_s_sg_sa_already_established** must be generated).
- If a **targetResultToken** is expected, the target generates its random number and concatenates it to the end on the initiator's random number. The concatenated value is then taken to be the identifier of the Security Association being established.

seq-number - This field must be present if replay detection or message sequencing have been specified as being required at Security Association initiation time. The field contains a message sequence number whose value is incremented by one for each message in a given direction, as specified by **directionIndicator**. The first message sent by the initiator following the **InitialContextToken** shall have the message sequence number specified in that token, or if this is missing, the value **0**. The first message returned by the target shall have the message sequence number specified in the **TargetReplyToken** if present, or failing this, the value **0**.

The receiver of the token will verify the sequence number field by comparing the sequence number with the expected sequence number and the direction indicator with the expected direction indicator. If the sequence number in the token is higher than the expected number, then the expected sequence number is adjusted and **GSS_S_GAP_TOKEN** is returned. If the token sequence number is lower than the expected number, then the expected sequence number is not adjusted and **GSS_S_DUPLICATE_TOKEN** or **GSS_S_OLD_TOKEN** is returned, whichever is appropriate. If the direction indicator is wrong, then the expected sequence number is not adjusted and **GSS_S_UNSEQ_TOKEN** is returned.

userData - See specific token type narratives below.

directionIndicator - FALSE indicates that the sender is the context initiator, TRUE indicates that the sender is the target.

pmtSeal - See specific token type narratives below.

15.13.16.1 MICToken

An **MICToken** is a per-message token, separate from the user data being protected, which can be used to verify the integrity of that data as received. The token is passed in the **message_protection_token** in SECIOP messages, and the protected data follows as a GIOP message or message fragment. The syntax of the token is:

```
MICToken ::= PMToken
```

The overall structure and field contents of the token are described above.

Fields specific to the **MICToken** are:

userData - Not present for **MICTokens**.

pmtSeal - The **Checksum** is calculated over the **DER** encoding of the **pmtContents** field with the **user data** temporarily placed in the **userData** field. The **userData** field is not transmitted.

15.13.16.2 WrapToken

A **WrapToken** encapsulates the input user data (optionally encrypted) along with associated integrity check values. It consists of an integrity header followed by a body portion that contains either the plaintext or encrypted data. The syntax of the token is:

```
WrapToken ::= PMToken
```

The overall structure and field contents of the token are described above.

Fields specific to the **WrapToken** are:

userData - Present either in plain text form or encrypted. If the data is encrypted, it is performed using the Confidentiality Dialogue Key, and as in [13], an 8-byte random confounder is first prepended to the data to compensate for the fact that an **IV** of **zero** is used for encryption.

wtSeal - The **Checksum** is calculated over the **pmtContents** field, including the **userData**. If the **userData** field is to be encrypted, the seal value is computed prior to the encryption.

15.13.17 ContextDeleteToken

The **ContextDeleteToken** is issued by either the context initiator or the target to indicate to the other party that the context is to be deleted.

```
ContextDeleteToken ::= SEQUENCE {
    cdtContents      [0]   CDTContents,
    cdtSeal          [1]   Seal
                  -- seal over cdtContents, encrypted under the Integrity
                  -- Dialogue Key. Contains only the sealValue field
}

CDTContents ::= SEQUENCE {
    tokenType        [0]   OCTET STRING VALUE X'0301',
```

```
        SAId            [1]   OCTET STRING,
        utcTime         [2]   UTCTime      OPTIONAL,
        usec            [3]   INTEGER      OPTIONAL,
        seq-number      [4]   INTEGER      OPTIONAL,
}
```

cdtContents - This contains only administrative fields, identifying the token type, the context, and providing exchange integrity.

seq-number - When present, this field contains a value one greater than that of the **seq-number** field of the last token issued from this issuer. The other administrative fields are as described above.

trtSeal - See above for a general description of the use of this construct.

15.13.18 Security Attributes

15.13.18.1 Data Structures

The security attribute is a basic construct for privilege and other attributes in PACs.

```
SecurityAttribute ::= SEQUENCE{
        attributeType           Identifier,
        attributeValue          SET OF SEQUENCE {
                                definingAuthority  [0] Identifier   OPTIONAL,
                                securityValue              [1] SecurityValue
        }
}

Identifier ::= CHOICE{
        objectId                [0]    OBJECT IDENTIFIER,
        directoryName           [1]    Name,
                -- imported from the Directory Standard
        printableName           [2]    PrintableString,
        octets                  [3]    OCTET STRING,
        intVal                  [4]    INTEGER,
        bits                    [5]    BIT STRING,
        pairedName              [6]    SEQUENCE{
                                       printableName    [0] PrintableString,
                                       uniqueName       [1] OCTET STRING
                                       }
}

SecurityValue ::= CHOICE{
        directoryName           [0]    Name,
        printableName           [1]    PrintableString,
        octets                  [2]    OCTET STRING,
        intVal                  [3]    INTEGER,
        bits                    [4]    BIT STRING,
        any                     [5]    ANY -- defined by attributeType
}
```

Only one set member is permitted in AttributeValue. Multivalue attributes are effected in the **securityValue** field, where the “SEQUENCE OF” construct can be used. (Including “SET OF” in the syntax enables security attributes to be stored as normal in a Directory whenever the choice made within Identifier is OBJECT IDENTIFIER.)

A directory name is translated into a string format as defined in Section 15.13.11, “Security Names,” on page 15-244. The **sequence<octet>** attribute value returned at the IDL interface is a representation of this string, not the more complex ASN.1 definition of this.

attributeType - Defines the type of the attribute. Attributes of the same type have the same semantics when used in Access Decision Functions, though they may have different defining authorities.

definingAuthority - The authority responsible for the definition of the semantics of the value of the security attribute. This optional field of the **attributeValue** can be used to resolve potential value clashes. It is defined as an Identifier which has a choice of syntax. For CSI-ECMA, it is always a **directoryName**.

securityValue - The value of the security attribute. Its syntax can be either one of the basic syntaxes for attributes or a more complex one determined by the attribute type.

15.13.18.2 Attribute Types

An attribute type in this standard is formally defined as an Identifier which provides a choice of syntax; however, all standard attribute types are defined as OBJECT IDENTIFIERs. Three types of attributes are defined:

1. Privilege attributes (e.g., **AccessId**, **GroupId**, **Role**)
2. Miscellaneous attributes, mainly the **AuditId**
3. Qualifier attributes used within the PV/CV delegation scheme to say where credentials can be used/delegated.

For standard attributes, the OBJECT IDENTIFIER includes:

- first, a standard part with the value **1.3.12.1.46**,
- then the “family” for privilege, miscellaneous, or qualifier attributes (**4**, **3**, or **5**), and
- then the value for that particular attribute type.

All standard attributes, which conformant ORBs must be able to generate/transmit, have this form.

In addition, conformant ORBs must be able to handle other attribute types defined in this chapter. They must also be able to handle attribute types with “OMG” object identifiers, as described in “Mapping Other Attributes to Externally Valid IDL Attributes” on page 15-199. In this case, the Object Identifier is:

<iso>..<omg>.<security><family definer>.<family>.<attribute type>

where the values of the CORBA family definer, CORBA family and attribute type are as defined in Appendix Section A.11.1, “Attribute Types,” on page 15-307. For standard attributes, the family definer is **0** and the family is **0** for privileges and **1** for miscellaneous attributes.

OMG Object Identifiers can also be used for privilege attributes defined by other organizations, who have registered a family definer with OMG.

15.13.19 Privilege and Miscellaneous Attribute Definitions

Privilege and miscellaneous attribute types are normally identified by Object Identifiers which have a standard part, then family and attribute type parts.

The following privilege and miscellaneous attributes are defined in the CORBA Security specification and have defined attribute types. Some of these are mandatory for a CSI level 2 conformant ORB to generate (see Section 15.8.15, “Support for CORBA Security Facilities and Extensibility,” on page 15-202). The Object Identifier in the privilege attribute set for that type is listed in Table 15-18 on page 15-255.

Table 15-18 Privilege and Miscellaneous Attributes

Type of Attribute	**oid family & type**	**Syntax**	**Meaning**
access-identity	4.2	printableString	The access identity represents the principal's identity to be used for access control purposes.
primary-group	4.3	printableString	The primary group represents a unique group to which a principal belongs. A security context must not contain more than one primary group for a given principal.
group	4.4	SEQUENCE OF printableString	A group represents a characteristic common to several principals. A PAC may contain more than one group for this principal.
role	4.1	printableString	A role attribute represents one of the principal's organizational responsibilities.
audit_id	3.2	printableString	The identity of the principal as used for auditing.

15.13.20 Qualifier Attributes

When a **targetQualification** or **delegateTargetQualification** method is present in the PAC, the syntax used for the method parameters is **securityAttribute**. Object Identifiers for qualifier attributes have the value **1.3.12.1.46.5**.<qualifier attribute type>.

Currently, only one form of qualifier attribute is defined, and this identifies the target by security name. This is usually the name of an identity domain as defined in “Domains” on page 15-38, not an individual object.

In the future, other forms of qualifier attributes may be added. For example, the attribute could identify an invocation delegation domain, rather than particular named target.

15.13.21 Target Names

Within a PAC protection method, a target name is indicated using the OID:

```
target-name-qualifier OBJECT IDENTIFIER ::= {qualifier-attribute 1 }
    Its syntax in the PAC is:
    TargetNameValueSyntax ::= Identifier
```

15.13.22 PAC Format

The **PAC** is in the form of a generalized certificate. A Generalized Certificate is composed of three main structural components:

1. The "**commonContents**" fields collectively serve to provide generally required management and control over the use of the PAC.
2. The "**specificContents**" fields are different for different types of certificate, and contain a type identifier to indicate the type. In this specification, only one type is defined - the Privilege Attribute Certificate (PAC).
3. The "**checkValue**" fields are used to guarantee the origin of the certificate. This is a signature in the CSI-ECMA specification (though a seal would be possible as in ECMA 235).

Common Certificate Contents	PAC-specific contents			Check Value
	protection/ delegation methods	privilege and other attributes	restrictions	

Figure 15-62 Generalized Certificate's Structural Components

```
GeneralizedCertificate ::= SEQUENCE{
    certificateBody     [0]  CertificateBody,
    checkValue          [1]  CheckValue
}

CertificateBody ::= CHOICE{
    encryptedBody       [0]  BIT STRING,
    normalBody          [1]  SEQUENCE{
                             commonContents      [0] CommonContents,
                             specificContents    [1] SpecificContents
                        }
}
```

The next sections describe these three main structural components of the Generalized Certificate.

15.13.23 Common Contents Fields

```
CommonContents ::= SEQUENCE{
        comConSyntaxVersion      [0]   INTEGER { version1 (1) }DEFAULT 1,
        issuerDomain             [1]   Identifier          OPTIONAL,
        issuerIdentity           [2]   Identifier,
        serialNumber             [3]   INTEGER,
        creationTime             [4]   UTCTime             OPTIONAL,
        validity                 [5]   Validity,
        algId                    [6]   AlgorithmIdentifier,
        hashAlgId                [7]   AlgorithmIdentifier      OPTIONAL
}
```

In the imported definition of **AlgorithmIdentifier**, ISO currently permits both a hash and a cryptographic algorithm to be specified. If this is done, they must appear in the algId field. The **hashAlgId** field is present for those cases where a separate hash algorithm specification is required.

```
Validity ::= SEQUENCE {
        notBefore        UTCTime,
        notAfter         UTCTime
} -- as in [ISO/IEC 9594-8]
 -- Note: Validity is not tagged, for compatibility with the
-- Directory Standard.
```

comConFieldsSyntaxVersion - Identifies the version of the syntax of the combination of the **commonContents** and the **checkValue** fields parts of the certificate.

issuerDomain - The security domain of the issuing authority. Not required if the form of issuerIdentity is a full distinguished name, but required if other forms of naming are in use. In CSI-ECMA, this is always a directoryName.

issuerIdentity - Identity of the issuing authority for the certificate.

serialNumber - Serial number of the certificate (**PAC**) as allocated by the issuing authority.

creationTime - The **UTCtime** that the certificate was created, according to the authority that created it.

validity - A pair of start and end times within which the certificate is deemed to be valid.

algId - The identifier of the secret or of the public cryptographic algorithm used to seal or to sign the certificate. If there is a single identifier for both the encryption algorithm and the hash function, it appears in this field.

hashAlgId - The identifier of the hash algorithm used in the seal or in the signature.

The certificate can be uniquely identified by a combination of the **issuerDomain**, **issuerIdentity**, and **serialNumber**.

15.13.24 Specific Certificate Contents for PACs

```
SpecificContents ::= CHOICE{
```

```
        pac                     [1]   PACSpecificContents
        -- only the PAC is used here
}

PACSpecificContents ::= SEQUENCE{
        pacSyntaxVersion        [0]   INTEGER{  version1 (1)}         DEFAULT 1,
        protectionMethods       [2]   SEQUENCE OF MethodGroup         OPTIONAL,
        pacType                 [4]   ENUMERATED{
                                      primaryPrincipal  (1),
                                      temperedSecPrincipal  (2),
                                      untemperedSecPrincipal(3)
                                }      DEFAULT 3,
        privileges              [5]   SEQUENCE OF PrivilegeAttribute,
        restrictions            [6]   SEQUENCE OF Restriction         OPTIONAL,
        miscellaneousAtts       [7]   SEQUENCE OF SecurityAttribute   OPTIONAL,
        timePeriods             [8]   TimePeriods                     OPTIONAL
}

PrivilegeAttribute ::= SecurityAttribute

Restriction ::= SEQUENCE {
        howDefined      [0] CHOICE {
                                included        [3] BIT STRING
                        },
                                -- the actual restriction in a form undefined here
        type            [2] ENUMERATED  {
                                mandatory (1),
                                optional    (2)
                        }       DEFAULT mandatory,
        targets         [3] SEQUENCE OF SecurityAttribute    OPTIONAL
}               -- applies to all targets if this is omitted
```

pacSyntaxVersion - The syntax version of the **PAC**.

protectionMethods - A sequence of optional groups of **Method** fields used to protect the certificate from being stolen or misused. For a full description see below.

pacType - Indicates whether the privileges contained in the **PAC** are those of a Primary Principal (e.g., the client) or of a Secondary Principal (e.g., the user). In this specification, it is always a **PAC** of a secondary principal untempered by the privileges of a Primary Principal.

privileges - Privilege Attributes of the principal.

restrictions - This field enables the original owner of the **PAC** to impose constraints on the operations for which it is valid. There are two types of restriction:

- Mandatory: If a target to which the restriction applies cannot understand the bit string defining the restriction, access should not be granted.
- Optional: If a target application to which the restriction applies cannot understand the bit string, it is expected to ignore it.

For CSI-ECMA, it is not mandatory to generate restrictions, but mandatory restrictions cannot be ignored. If not understood, the **PAC** cannot be accepted.

miscellaneousAtts - Security attributes which are neither privileges attributes nor restrictions attributes. In a **PAC**, this may include identity attributes such as Audit Identity. For the CSI-ECMA specification, this is the only miscellaneous attribute expected.

timePeriods - This field adds further time restrictions to the validity field of the **commonContents**. Either **startTime** or **endTime** can be optional. The **TimePeriods** control is passed if:

- the time now is within any of the sequence periods, or
- if there is a period with a start before now and no **endTime**, or
- there is a period with an end after now and no **startTime**.

15.13.24.1 Protection Methods

A method consists of a method id and parameters (**methodParams**). The method id determines the syntax for the type of **methodParams**.

```
Method ::= SEQUENCE{
        methodId        [0]   MethodId,
        methodParams    [1]   SEQUENCE OF Mparm     OPTIONAL
}
MethodId ::= CHOICE{
        predefinedMethod      [0] ENUMERATED {
                                  controlProtectionValues    (1),
                                  ppQualification            (2),
                                  targetQualification        (3),
                                  delegateTargetQualification  (4)
                              }
}

Mparm ::= CHOICE{
        pValue                [0]    PValue,
        securityAttribute     [1]    SecurityAttribute
}

PValue ::= SEQUENCE{
        pv                    [0]    BIT STRING
        algorithmIdentifier   [1]    AlgorithmIdentifier    OPTIONAL
}

CertandECV ::=   SEQUENCE {
        certificate           [0]    GeneralizedCertificate,
        ecv                   [1]    ECV        OPTIONAL
}
        - ECV is defined in later
```

methodId - Identifies a protection method. Methods can be used in any combination, and except where stated otherwise, multiple occurrences of the same method are permitted. The choice of methodId determines the permitted choices of method parameters in the methodParams construct as described below.

methodParams - Parameters for a protection method. The semantics of each protection method is described in section Section 15.8.9.2, "Cryptographic Profiles," on page 15-186.

For the Primary Principal Qualification Method, the **MethodId** is **ppQualification** and the syntax of **Mparm** is **securityAttribute**. Its value is defined in Section 15.13.8, "PPID Method," on page 15-243.

For the PV/CV method, the **MethodId** is: **controlProtectionValues** and the syntax of Mparm is: **pValue**.

For the Target Qualification protection method, the **MethodId** is **targetQualification** and the syntax for **Mparms** is **securityAttribute**.

For the Delegate/Target Qualification protection method, the **MethodId** is **delegatetargetQualification** and the syntax for **Mparms** is **securityAttribute**.

The security attribute in the target and delegate/target protection method is a qualifier attribute as defined in Section 15.13.20, "Qualifier Attributes," on page 15-255.

15.13.24.2 *External Control Values Construct*

When using the **controlProtectionValues** method a **`PAC`-**protected under that method may be accompanied by one or more control values and indices to the method occurrences in the certificate to which they apply. Also, when such a certificate is being issued to a requesting client, the **CV** values it will need in order to use that certificate may need to be returned with it.

```
ECV ::= SEQUENCE {
    crypAlgIdentifier    [0]  AlgorithmIdentifier   OPTIONAL,
    cValues              [1]  CHOICE {
                              encryptedCvalueList     [0] BIT STRING,
                              individualCvalues       [1] CValues
                         }
}

CValues ::= SEQUENCE OF SEQUENCE {
    index                [0]  INTEGER,
    value                [1]  BIT STRING
}
```

crypAlgIdentifier - Specifies the encryption algorithm of the control values.

cValues - An **ECV** construct can contain either an encrypted list of control values in the **encryptedCvalueList** field, or a list of individual control values in **individualCvalues**.

If the **encryptedCvalueList** choice is made, the whole list is encrypted in bulk, but the in-clear contents of this field are expected to have the syntax **CValues**. If the **individualCvalues** choice is made, values are individually encrypted in the value fields of the list. Encryption is always done under the basic key protecting the operation.

In the case of the **controlProtectionValues** method, value is a **CV**, and index is then the index of the method occurrence in the certificate, starting at 1.

15.13.25 Check Value

In this specification, a **PAC** is protected by being digitally signed by the issuer.

A signature may be accompanied by information identifying the Certification Authority under which the signature can be verified, and by an optional convenient reference to, or the actual value of, the user certificate for the private key that the signing authority used to sign the certificate.

```
CheckValue ::= CHOICE{
      signature                    [0]  Signature
      -- only signature supported here
}

Signature ::= SEQUENCE{
      signatureValue               [0]   BIT STRING,
      publicAlgId                  [1]   AlgorithmIdentifier   OPTIONAL,
      hashAlgId                    [2]   AlgorithmIdentifier   OPTIONAL,
      issuerCAName                 [3]   Identifier   OPTIONAL,
      caCertInformation            [4]   CHOICE {
                                         caCertSerialNumber     [0]   INTEGER,
                                         certificationPath      [1]   CertificationPath
                                   } OPTIONAL
}
--CertificationPath is imported from [22]
```

signatureValue - The value of the signature. It is the result of a public encryption of a hash value of the **certificateBody**.

publicAlgId - Only present if the certificate body is encrypted, then it is a duplication of the **algId** value in "**commonContents**." This is not required in CSI-ECMA.

hashAlgId - Only present if the certificate body is encrypted, then it is a duplication of the **hashAlgId** value in “**commonContents**.” This is not required in CSI-ECMA.

issuerCAName - The identity of the Certification Authority that has signed the user certificate corresponding to the private key used to sign this certificate.

caCertInformation - Contains either just a certificate serial number (which together with the **issuerCAName** uniquely identifies the user certificate corresponding to the private key used to sign this certificate), or a full specification of a certification path via which the validity of the signature can be verified. The latter option follows the approach used in [22].

The **Seal** structure is used in the **Tokens** defined above.

```
Seal ::= SEQUENCE{
                         sealValue          [0]   BIT STRING,
                         secretAlgId        [1]   AlgorithmIdentifierOPTIONAL,
                         hashAlgId          [2]   AlgorithmIdentifierOPTIONAL,
                         targetName         [3]   Identifier OPTIONAL,
                         keyId              [4]   INTEGEROPTIONAL
}
```

sealValue - The value of the seal. It is the result of a secret encryption of a hash value of a set of octets (which are the **DER** encoding of some ASN.1 type)

secretAlgId - An optional indicator of the sealing algorithm.

hashAlgId - Only present if the **secretAlgId** does not specify which hashing algorithm is used.

targetName - Identifies the **targetAEF** or target with which the secret key used for the seal is shared.

keyId - This serial number together with the **targetName** uniquely identifies the secret key used in the seal.

15.13.26 Basic Key Distribution

The **TargetKeyBlock** is structured as follows:

- An identifier (**kdSchemeOID**) for the key distribution scheme being used, which takes the form of an OBJECT IDENTIFIER.
- A part which, if present, the target AEF needs to pass on to its KDS (**targetKDSPart** - will be present only when the target AEF's KDS is different from the initiator's).
- A part which, if present, can be used directly by the **targetAEF (targetPart**).

When a **targetAEF** using a separate KDS receives the **targetKeyBlock**, it first checks whether it supports the key distribution scheme indicated in **kdsSchemeOID**. Two different cases need to be considered:

1. Only the **targetPart** is present. The target AEF computes the basic key directly, using the information present in the **targetPart**. The syntax of **targetPart** is scheme-dependent. Expiry information optionally can be present in **targetPart**. If supported by the scheme, the Primary Principal attributes of the initiator will also be present for PAC protection under the Primary Principal Qualification method (see above).
2. Only the **targetKDSPart** is present. The **targetAEF** forwards the **TargetKeyBlock** to its KDS. In return, it receives a scheme-dependent data structure which allows the target AEF to determine the basic key and, if supported by the scheme, the Primary Principal attributes of the initiator for PAC protection purposes. Expiry information can optionally be present in the **targetKDSPart**.

The form of this information depends on the key distribution configuration in place.

15.13.27 Keying Information Syntax

```
TargetKeyBlock ::= SEQUENCE {
        kdSchemeOID          [2]   OBJECT IDENTIFIER,
        targetKDSpart        [3]   ANY         OPTIONAL,
                                         -- depending on kdSchemeOID
        targetPart           [4]   ANY         OPTIONAL
                                         -- depending on kdSchemeOID
}
```

kdSchemeOID - Identifies the key distribution scheme used. Allows the **targetAEF** to determine rapidly whether or not the scheme is supported. It also allows for the easy addition of future schemes.

targetKDSpart - Part of the Target Key Block which is processable only by the **KDS** of the target **AEF**. This part is sent by the target **AEF** to its local **KDS**, in order to get the basic key which is in it. It must always contain the name of a target "served" by the **targetAEF** in question. The mapping between the name of the application and the name of the target AEF is known to the target **AEF**'s **KDS** which is able to authenticate which **targetAEF** is issuing the request for translating the **targetKDSpart**. It can then verify that the AEF is one which is responsible for the application name contained in the **targetKDSpart**. If it is, the key is released and is sent protected back to the requesting **AEF**. **TargetKDSpart** should include data that enables the **KDS** of the target **AEF** to authenticate the **KDS** of the initiator. When the "Primary Principal Qualification" protection method needs to be used for the **PAC**, unless there is an accompanying **targetPart**, **targetKDSpart** must contain the appropriate primary principal security attributes (which is always true in this specification).

targetPart - A part of the Target Key Block which is processed only by the target **AEF**. When there is no **targetKDSpart** it is processable directly; otherwise, it can only be processed after the target **KDSpart** has been processed by the **KDS** of the target **AEF**, and the appropriate Keying Information has been returned to the **AEF**. The targetPart construct should include data that enables the target **AEF** to authenticate the **KDS** of the initiator. When the "Primary Principal Qualification" protection method needs to be used for the **PAC**, **targetPart** must contain the primary principal security attributes.

15.13.28 Summary of Key Distribution Schemes

This specification defines three key distribution schemes. These are:

1. **symmIntradomain**: using a secret key technology within a domain. In this case, the **targetKDSpart** of the **TargetKeyBlock** is not supplied and the **targetPart** contains a Kerberos ticket.
2. **hybridInterdomain**: In this case, the **targetPart** field is not supplied. The **PublicTicket** contains a Kerberos ticket.
3. asymmetric: the **targetKDSpart** is not supplied and the **targetPart** contains an **SPKM_REQ**.

Table 15-19 shows the different syntaxes used for **targetKDSpart** and **targetPart** for the defined KD-schemes. “Missing” in the table means that the relevant construct is not supplied.

Table 15-19 Syntaxes Used for targetKDSpart and targetPart

KD-Scheme name	kdSchemeOID	targetKDSpart	targetPart
symmIntradomain	{kd-schemes 1}	Missing	Ticket
hybridInterdomain	{kd-schemes 3}	PublicTicket	Missing
asymmetric	{kd-schemes 6}	Missing	SPKM_REQ

Further options are possible by defining further kd-schemes. For example, ECMA 235 also defines options for:

- initiators with public keys and targets with secret keys
- initiators with secret keys and targets with public keys

15.13.29 CSI-ECMA Secret Key Mechanism

In this scheme, the client and target each share different secret keys with the same Key Distribution Server.

To establish the association between the client and target the client obtains a **targetKeyBlock** from its **KDS** containing a basic key encrypted under the target’s long term key. On receipt of the targetKeyBlock, the target can extract the basic key from it.

The *symmIntradomain* key distribution scheme:

- has a mechanism id of **CSI_ECMA_Secret**, and
- uses a Kerberos ticket in the **targetKeyBlock** of the **initial_context_token**. An unmodified Kerberos **TGS** can be used as the **KDS** in this case.

15.13.29.1 Profile of Ticket as Used in SymmIntradomain Scheme

Table 15-20 indicates which optional fields must be present in the Kerberos ticket for the **CSI_ECMA_Secret** mechanism and indicates the values which are required to be present in all fields.

Table 15-20 Kerberos Ticket’s Mechanism Fields

Field	Value/Constraint
tkt-vno	5
realm	ticket issuer's domain name in Kerberos realm name form
sname	target application name including the realm of the target
- EncTicketPart	encrypted with long term key of target AEF

Table 15-20 Kerberos Ticket’s Mechanism Fields *(Continued)*

Field	**Value/Constraint**
-- flags	only bits 6, 10 and 11 can be meaningful in the context of the CSI-ECMA protocol, the rest are ignored
-- key	the basic key
-- crealm	initiator domain name in Kerberos realm name form
-- cname	principal name of the initiator (in the case of delegation the cname will be that of the delegate)
-- transited	not used
-- authtime	the time at which the initiator was authenticated
-- starttime	not used
-- endtime	the time at which the ticket becomes invalid
-- renew-till	not used
-- caddr	not used
-- authorization-data	contains the PPID corresponding to cname

The Kerberos Ticket's **authorization_data** field contains the **PPID** of the context initiator, as formally defined below.

```
ECMA-AUTHORIZATION-DATA-TYPE ::= INTEGER { ECMA-ADATA (65) }
ECMA-AUTHORIZATION-DATA ::= SEQUENCE {
        ecma-ad-type      [0]  ENUMERATED {ppidType (0)},
        ecma-ad-value     [1]  CHOICE {ppidValue [0]   SecurityAttribute

}
```

ppidType - Indicates the type of the authorization data which is included in the **Ticket**.

ppidValue - This value is used in the **ppQualification PAC** protection method, as described above.

15.13.30 CSI-ECMA Hybrid Mechanism

In this scheme, the initiator shares a secret key with its **KDS** and the target shares a secret key with another **KDS**. In addition, each **KDS** possesses a private/public key pair.

To establish the client-target association, the client gets a **targetKeyBlock** from its **KDS** containing the basic key encrypted under a temporary key and the temporary key encrypted under the target’s **KDS** public key. The **targetKeyBlock** is also signed using the initiator’s **KDS** private key.

On receipt of the **targetKeyBlock**, the target transmits it to its **KDS** and gets back the basic key encrypted under the long term secret key it shares with its **KDS**.

The **hybridInterdomain** key distribution scheme

- has a mechanism id of **CSI_ECMA_Hybrid** in the IOR, and
- uses a Public ticket in the **targetKeyBlock** of the **initial_context_token**, as described below. A modified Kerberos **TGS** can be used as the **KDS** in this case.

15.13.30.1 Hybrid Inter-domain Key Distribution Scheme Data Elements

```
PublicTicket ::= SEQUENCE{
        krb5Ticke        [0]  Ticket,
        publicKeyBlock   [1]  PublicKeyBlock
    }

    PublicKeyBlock ::= SEQUENCE{
        signedPKBPart           [0]   SignedPKBPart,
        signature               [1]   Signature          OPTIONAL,
        certificate             [2]   Certificate        OPTIONAL
    }

    SignedPKBPart ::= SEQUENCE{
        keyEstablishmentData    [0]   KeyEstablishmentData,
        encryptionMethod        [1]   AlgorithmIdentifier  OPTIONAL,
        issuingKDS              [2]   Identifier,
        uniqueNumber            [3]   UniqueNumber,
        validityTime            [4]   TimePeriods,
        creationTime            [5]   UTCTime
    }

    UniqueNumber ::= SEQUENCE{
        timeStamp               [0]   UTCTime,
        random                  [1]   BIT STRING
    }
```

krb5Ticket - The Kerberos Ticket which contains the basic key. The encrypted part of this ticket is encrypted using the key found within the **encryptedPlainKey** field of the **KeyEstablishmentData** in the **PublicKeyBlock**.

publicKeyBlock - Contains the key used to protect the **krb5Ticket** encrypted using the public key of the recipient and signed by the encryptor (i.e., the context initiator's KD-Server).

signedPKBPart - The part of the **publicKeyBlock** which is signed. The **keyEstablishmentData** field contains the **KeyEstablishmentData** (i.e., the actual encrypted temporary key).

- The **encryptionMethod** indicates the algorithm used to encrypt the **encryptedKey**.
- The **issuingKDS** is the name of the KD-Server which produced the **PublicTicket**.
- The **uniqueNumber** is a value (containing a timestamp and a random number) which prevents replay of the **PublicTicket**.
- **validityTime** specifies the times for which the **PublicTicket** is valid.
- creationTime contains the time at which the **PublicTicket** was created.

signature - Contains the signature calculated by the **issuingKDS** on the **signedPKBPart** field.

certificate - If present, this contains the public key certificate of the issuing **KDS**.

15.13.30.2 Key Establishment Data Elements

These are used in public key establishment mechanisms.

```
KeyEstablishmentData ::= SEQUENCE {
        encryptedPlainKey    [0]   BIT STRING,-- encrypted PlainKey
        targetName           [1]   Identifier   OPTIONAL,
        nameHashingAlg       [2]   AlgorithmIdentifier        OPTIONAL
    }

    HashedNameInput ::= SEQUENCE {
        hniPlainKey          [0]  BIT STRING,-- same as plainKey
        hniIssuingKDS        [1]  Identifier


    PlainKey ::= SEQUENCE {
        plainKey             [0]  BIT STRING,  -- The cleartext key
        hashedName           [1]  BIT STRING
    }
```

encryptedPlainKey - Contains the encrypted key. The BIT STRING contains the result of encrypting a **PlainKey** structure.

targetName - If present, contains the name of the target application. This is necessary for some of the KD-schemes.

nameHashingAlg - Specifies the algorithm which is used to calculate the **hashedName** field of the **PlainKey**.

hniPlainKey
hniIssuingKDS - Used as input to a hashing algorithm as a general means to prevent ciphertext stealing attacks.

plainKey - Contains the actual bits of the plaintext key which is to be established.

hashedName - A hash of the name of the encrypting **KDS** calculated using the plainkey and **KDS** name as input (within the **HashedNameInput** structure). The algorithm identified in **nameHashingAlg** is used to calculate this value.

targetName - If present, contains the name of the target for which the **PublicTicket** was originally produced. This may be different from the targetIdentity field of the **initialContextToken** if caching of **PublicTickets** has been implemented.

15.13.30.3 Key Establishment Algorithm

The **PublicKeyBlock** in this mechanism and the **SPKM_REQ** construct used in scheme 6 requires a sequence of key establishment algorithm identifier values to be inserted into the **key_estb_set** field. The OBJECT IDENTIFIER below is defined as the (single) key establishment "algorithm" for ECMA mechanisms:

```
gss-key-estb-alg AlgorithmIdentifier ::= {kd-schemes, NULL }
```

gss-key-estb-alg - Identifies the key establishment algorithm value to be used within the **key_estb_set** field of an **SPKM_REQ** data element as the one defined by ECMA.

This algorithm is used to establish a symmetric key for use by both the initiator and the target AEF as part of the context establishment. The corresponding **key_estb_req** field of the **SPKM_REQ** will be a BIT STRING the content of which is a **DER** encoding of the **KeyEstablishmentData** element.

15.13.30.4 Profile of Ticket as Used in Hybrid Interdomain Scheme

Note that the **krb5Ticket** part of this is identical to that used in the **CSI_ECMA_Secret** key mechanism except that the **EncTicketPart** is encrypted with the temporary key used between **KDS** rather than the target's key.

Table 15-21 Ticket as Used in Hybrid Interdomain Scheme

Field	Value/Constraint
krb5Ticket	
- tkt-vno	5
- realm	initiator domain name in Kerberos realm name form
- sname	target application name including the realm of the target
-- EncTicketPart	encrypted with temporary key (which is in turn encrypted within the keyEstablishmentData field)
--- flags	only bits 6, 10, and 11 can be meaningful in the context of the CSI-ECMA protocol, the rest are ignored
--- key	the basic key
--- crealm	initiator domain name in Kerberos realm name form
--- cname	principal name of the initiator (in the case of delegation the cname will be that of the delegate)
--- transited	not used
--- authtime	the time at which the initiator was authenticated
--- starttime	not used
--- endtime	the time at which the ticket becomes invalid

Table 15-21 Ticket as Used in Hybrid Interdomain Scheme *(Continued)*

Field	Value/Constraint
--- renew-till	not used
--- caddr	not used
--- authorization-data	contains the PPID corresponding to cname
publicKeyBlock	
- signedPKBPart	
-- encryptedKey	KeyEstablishmentData structure
-- encryptionMethod	gss-key-estb-alg
-- issuingKDS	X.500 name of initiator's KDS (the signer)
-- uniqueNumber	creation time of publicKeyBlock plus a random bit string
-- validityTime	only one period allowed
-- creationTime	creation time of publicKeyBlock
- signature	contains all the signing information as well as the actual signature bits
- certificate	optional

15.13.31 CSI-ECMA Public Mechanism

In this scheme, both client and target possess a private/public key pair and neither use a **KDS**.

To establish the client-target association, the client constructs a **targetKeyBlock** containing a basic key encrypted under the target's public key. The target key block is signed with the client's private key. On receipt of the **targetKeyBlock**, the target directly establishes a basic key from it.

The **asymmetric** key distribution scheme:

- has a mechanism id of **CSI_ECMA_Public**, and
- uses an **SPKM_REQ** in the **targetKeyBlock** of the **initial_context_token**.

This mechanism has only a profile of the **SPKM_REQ** as defined below.

15.13.31.1 Profile of SPKM_REQ Used in Public Key Mechanism

Table 15-22 indicates which optional fields must be present in the **SPKM_REQ** in the **targetKeyBlock** for the **CSI_ECMA_Public** mechanism and indicates the values which are required to be present in all fields.

Table 15-22 SPKM-REQ Used in Public Key Mechanism

Field	Value/Constraint
requestToken	
- tok_id	not used - fixed value of ‘0'
- context_id	not used - fixed value of bit string containing one zero bit
- pvno	not used - fixed value of bit string containing one zero bit
- timestamp	creation time of SPKM_REQ - required
- randSrc	random bit string
- targ_name	X.500 Name of target AEF
- src_name	X.500 Name of initiator
- req_data	
-- channelId	not used - octet string of length one value ‘00'H
-- seq_number	missing
-- options	not used - all bits set to zero
-- conf_alg	not used - use NULL CHOICE
-- intg_alg	not used - use a SEQUENCE OF with zero elements
- validity	mandatory
- key_estb_set	only one element supplied containing gss-key-estb-alg
- key_estb_req	contains KeyEstablishmentData with targetApplication field missing
- key_src_bind	missing
req_integrity	sig_integ mandatory
certif_data	only userCertificate field supported
auth_data	missing

Definitions of **KeyEstablishmentData** and **gss-key-estb-alg** are given in Section 15.13.30, “CSI-ECMA Hybrid Mechanism,” on page 15-265.

15.13.32 Dialogue Key Block

Dialogue Key Block constructs are used to specify how the integrity dialogue key and confidentiality dialogue key should be derived from the basic key, and specify the cryptographic algorithms with which the keys should be used. Dialogue keys are explained above. The syntax is as follows:

```
DialogueKeyBlock   ::=  SEQUENCE {
        integKeySeed                [0]   SeedValue,
        confKeySeed                 [1]   SeedValue,
        integKeyDerivationInfo      [2]   KeyDerivationInfo     OPTIONAL,
        confKeyDerivationInfo       [3]   KeyDerivationInfo     OPTIONAL,
        integDKuseInfo              [4]   DKuseInfo             OPTIONAL,
        confDKuseInfo               [5]   DKuseInfo             OPTIONAL
}

SeedValue  ::= SEQUENCE {
        timeStamp                   [0]   UTCTime               OPTIONAL,
        random                      [1]   BIT STRING
}

KeyDerivationInfo::= SEQUENCE {
        owfId                       [0]   AlgorithmIdentifier,
        keySize                     [1]   INTEGER
}

DKuseInfo    ::= SEQUENCE {
        useAlgId                    [0]   AlgorithmIdentifier,
        useHashAlgId                [1]   AlgorithmIdentifier   OPTIONAL
}
```

integKeySeed - A random number, optionally concatenated with a time value to ensure uniqueness, used as input to the one way function specified in **integKeyDerivationInfo**.

confKeySeed - A random number, optionally concatenated with a time value to ensure uniqueness, used as input to the one way function specified in **confKeyDerivationInfo**.

integKeyDerivationInfo - Key derivation information for the integrity dialogue key, as follows:

- **owfId -** The one way algorithm which takes the basic key XOR the seed as input, resulting in the integrity dialogue key.
- **keySize -** The size of the key in bits. If the algorithm identified by owfId produces a larger key, it is reduced by masking to this length, losing its most significant end.

confKeyDerivationInfo - Key derivation information for the confidentiality dialogue key. The fields in this construct have the same meanings as defined above for the integrity dialogue key.

integDKuseInfo - Information describing how the integrity dialogue key is to be used, as follows:

- **useAlgId -** The secret or public reversible encryption algorithm with which the integrity dialogue key is to be used.
- **useHashAlgId -** The one way function with which the integrity dialogue key is to be used. It is the hash produced by this algorithm on the data to be protected which is encrypted using **useAlgId**.

confDKuseInfo - Information describing how the confidentiality key is to be used. The **useHashAlgId** construct is not used here.

15.14 Integrating SSL with CORBA Security

15.14.1 Introduction

This section defines how SSL [21] is integrated with CORBA Security. SSL provides CSI level 0(see Appendix Section C.7.2, "Common Secure Interoperability Levels," on page 15-326) functionality only. This level of functionality is achieved only if the optional authentication features of SSL are used.

15.14.2 Cryptographic Profiles

All of the cryptographic profiles defined by SSL may be used by ORBs using SSL for Security.

15.14.3 IOR Encoding

A new kind of security tag is defined, for use in the component tag sequence in the IIOP IOR profile body, to describe the use of Secure Transports with CORBA Security. This enables the future use of combinations of security mechanisms and secure transports.

The IIOP TAG identifying the SSL-secure transport is **TAG_SSL_SEC_TRANS**. The tag component data described below must be encapsulated using CDR encoding. The data structure associated with this tag is as follows:

```
struct SSL {
    Security::AssociationOptions      target_supports;
    Security::AssociationOptions      target_requires;
    unsigned short                    port;
};
```

The definition of association options is the same as for the CSI protocols. SSL only supports client and target authentication if the optional certificate exchanger features of SSL are supported.

Unlike the CSI mechanism TAGs, the SSL TAG does not include cryptographic profiles, as cryptography is negotiated, as part of the SSL session establishment. For the same reason the TAG does not include a security name for the target.

The **port** field contains the port number to be used instead of the port defined in the accompanying IIOP profile body, if SSL is selected by the client. It contains the TCP/IP port number (at the specified host) where the target agent is listening for connection requests. The agent must be ready to process IIOP messages on connections accepted at this port.

As with the other secure interoperability options, if the client invokes the target without the appropriate level of security (e.g., if the client is not secure and simply invokes the target ignoring all security TAGs in the profile) the target shall raise the CORBA::NO_PERMISSION exception.

15.14.4 Relation to SECIOP

As SSL provides a secure transport layer over TCP/IP, the CORBA SECIOP protocol is not required when using SSL. Instead, the connection rules of IIOP (see the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*) are applied to SSL (which itself uses TCP).

15.15 DCE-CIOP with Security

This section describes how to provide secure interoperability between ORBs which use the DCE Common Inter-ORB Protocol (DCE-CIOP). It describes how the DCE-CIOP transport layer should handle security (for example, how it should interpret the security components of the IOR profile when selecting DCE Security Services for a request and secure invocation).

15.15.1 Goals of Secure DCE-CIOP

The original goals of DCE-CIOP, documented in the *Common Object Request Broker: Architecture and Specification*, are maintained and enhanced by Secure DCE-CIOP:

- Support multi-vendor, mission critical, enterprise-wide, secure ORB-based applications.
- Leverage services provided by DCE wherever appropriate.
- Allow efficient and straightforward implementation using public DCE APIs.
- Preserve ORB implementation freedom.

Secure DCE-CIOP achieves these goals by taking advantage of the integrated security services provided by DCE Authenticated RPC. It is not a goal of the Secure DCE-CIOP specification to support the use of arbitrary security mechanisms for protection of DCE-CIOP messages.

15.15.2 Secure DCE-CIOP Overview

Secure interoperability between ORBs using the DCE-CIOP transport relies on the DCE Security Services and the DCE Authenticated RPC runtime that utilizes those services.

The DCE Security Services (specified in [6]), as employed by the DCE Authenticated RPC runtime (specified in [7] and [8]), provide the following security features:

- cryptographically-secured mutual authentication of a client and target,
- ability to pass client identity and authorization credentials to the target as part of a request,
- protection against undetected, unauthorized modification of request data,
- cryptographic privacy of data, and
- protection against replay of requests and data.

The RPC runtime provides the communication conduit for exchanging security credentials between communicating parties. It protects its communications from threats such as message replay, message modification, and eavesdropping.

The DCE-CIOP uses DCE RPC APIs to request security features for a given client-target communication binding. Subsequent DCE-CIOP messages on that binding flow over RPC and thus are protected at the requested levels.

This Secure DCE-CIOP specification defines the IOR Profile components required to support Secure DCE-CIOP. Each component is identified by a unique tag, and the encoding and semantics of the associated **component_data** are specified. Client-secure association requirements, as indicated by client-side policy, and target-secure association requirements, as specified in the target IOR Profile security components, are mapped to DCE Security Services. Finally, the use of DCE APIs to protect DCE-CIOP messages is described.

15.15.2.1 IOR Security Components for DCE-CIOP

The information necessary to invoke secure operations on objects using DCE-CIOP is encoded in an IOR in a profile identified by **TAG_MULTIPLE_COMPONENTS**. The **profile_data** for this profile is a CDR encapsulation (see “CDR Transfer Syntax” in the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*) of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures. These types are described in the ORB Interoperability Architecture chapter of the *Common Object Request Broker: Architecture and Specification*.

The Multiple Component Profile contains the tagged components required to support DCE-CIOP, described in the DCE ESIOP chapter of the *Common Object Request Broker: Architecture and Specification*, as well as the components required to support security for DCE-CIOP. The general security components are described in “Security Components of the IOR” on page 15-179. The DCE-specific security component and semantics for the common security components are described here.

Although a conforming implementation of Secure DCE-CIOP is only required to generate and recognize the components defined here and in the General Inter-ORB Protocol chapter of the *Common Object Request Broker: Architecture and Specification*, the profile may also contain components used by other kinds of ORB transports and services. Implementations should be prepared to encounter profiles identified by **TAG_MULTIPLE_COMPONENTS** that do not support DCE-CIOP. Unrecognized components should be preserved but ignored. Although an implementation may choose to order the components in a profile in a particular way, other implementations are not required to preserve that order. Implementations must be prepared to handle profiles whose components appear in any order.

TAG_DCE_SEC_MECH

For a profile to support Secure DCE-CIOP, it must include exactly one **TAG_DCE_SEC_MECH** component. Presence of this component indicates support for the (non-GSSAPI) "DCE Security with Kerberos V5 with DES" mechanism type. The **component_data** field contains an authorization service identifier and an optional sequence of tagged components.

Future versions of DCE Security that require different information than what is provided by the **component_data** structure described below are expected to be supported with a new component tag, rather than with revisions to the data structure associated with the **TAG_DCE_SEC_MECH** tag.

The DCE Security Mechanism component is defined by the following OMG IDL:

```
module DCE_CIOPSecurity {

    const IOP::ComponentId TAG_DCE_SEC_MECH = 103

    // CORBA IDL doesn't (yet) support const octet
    //
    // const octet DCEAuthorizationNone = 0;
    // const octet DCEAuthorizationName = 1;
    // const octet DCEAuthorizationDCE = 2;

    typedef unsigned short     DCEAuthorization;

    const DCEAuthorization     DCEAuthorizationNone = 0;
    const DCEAuthorization     DCEAuthorizationName = 1;
    const DCEAuthorization     DCEAuthorizationDCE = 2;

    // since consts of type octet are not allowed in IDL the constant
    // values that can be assigned to the authorization_service field
    // in the DCESecurityMechanismInfo is declared as unsigned shorts.
    // when they actually get assigned to the authorization_service field
    // they should be assigned as octets.

    struct DCESecurityMechanismInfo {
        octet                          authorization_service;
        sequence <TaggedComponent>     components;
    };
};
```

A **TaggedComponent** structure is built for the DCE Security Mechanism component by setting the tag member to **TAG_DCE_SEC_MECH**, and setting the **component_data** member to a CDR encapsulation of a **DCESecurityMechanismInfo** structure.

The authorization_service Field

The **authorization_service** field is used to indicate what authorization service is required by the target, and therefore must be supported by the authenticated RPC runtime for invocations on this IOR. Two authorization models are supported: **DCEAuthorizationName** and **DCEAuthorizationDCE** with a third identifier, **DCEAuthorizationNone**, to indicate that no authorization is required.

The components Field

The **components** field contains a sequence of zero or more tagged components, none of which may appear more than once, from the following list of common security IOR components: **TAG_ASSOCIATION_OPTIONS** and **TAG_SEC_NAME**.

Each of these components, defined in “Security Components of the IOR” on page 15-179, may be present either in the components field of the **DCESecurityMechanismInfo** structure, or at the top level of the IOR profile. When one of these components appears at the top level of the profile, its data may be shared by other security mechanisms in the profile. When it appears in the nested components field of **DCESecurityMechanismInfo**, its data is available only to the DCE Security mechanism and overrides the data of an identically-tagged component, if present, at the top level of the profile.

15.15.2.2 *TAG_ASSOCIATION_OPTIONS*

The association options component, described in “Security Components of the IOR” on page 15-179, contains flags indicating which protection and authentication services the target supports and which it requires. This component is optional for Secure DCE-CIOP; defaults are used when the component is not present.

The way in which association options are interpreted for use with DCE security is reflected in Table 15-23 shows how an association option is mapped to a DCE RPC protection level and authentication service.

Table 15-23 Association Option Mapping to DCE Security

Association Option	**DCE RPC Protection Level**	**DCE RPC Authentication Service**
NoProtection	rpc_c_protect_level_none	rpc_c_authn_none
Integrity	rpc_c_protect_level_pkt_integrity	rpc_c_authn_dce_secret
Confidentiality	rpc_c_protect_level_pkt_privacy	rpc_c_authn_dce_secret
DetectReplay	rpc_c_protect_level_pkt	rpc_c_authn_dce_secret

Table 15-23 Association Option Mapping to DCE Security

Association Option	DCE RPC Protection Level	DCE RPC Authentication Service
DetectMisordering	rpc_c_protect_level_pkt	rpc_c_authn_dce_secret
EstablishTrustInTarget	rpc_c_protect_level_connect	rpc_c_authn_dce_secret
EstablishTrustInClient	rpc_c_protect_level_connect	rpc_c_authn_dce_secret
tag not present	rpc_c_protect_level_default	rpc_c_authn_dce_secret

If the **TAG_ASSOCIATION_OPTIONS** component is not present, then the target is assumed both to support and to require **rpc_c_protect_level_default** and **rpc_c_authn_dce_secret**. (The value of **rpc_c_protect_level_default** is defined by the DCE implementation or by a site administrator.)

The target_supports Field

When an association option is set in the **target_supports** field of the **TAG_ASSOCIATION_OPTIONS component_data**, it indicates that the target supports invocations which use Secure DCE-CIOP with the protection level and authentication service that correspond to the selected option, as shown in Table 15-23. Any or all of the association options may be set in the **target_supports** field. The options set in the **target_supports** field will be compared with client-side policy required options to determine if the target can support the client's requirements.

Although, for the DCE security mechanism, a single selected option may imply support for several other options (e.g., selection of the Integrity option implies support for **DetectReplay**, **DetectMisordering**, and **EstablishTrustInClient**) it is recommended that every supported option be explicitly set in the **target_supports** field to facilitate comparison with client requirements.

The target_requires Field

When an association option is set in the **target_requires** field of the **TAG_ASSOCIATION_OPTIONS component_data**, it indicates that the target requires invocations secured with at least the protection level and authentication service that correspond to the selected option, as shown in Table 15-23. Since DCE RPC supports a range of protection levels, each of which provides all the protection of the level below it and also some additional protection, selecting multiple **target_requires** options does not make sense. For DCE, no more than one option need be selected in the **target_requires** field.

If a **TAG_ASSOCIATION_OPTIONS** component is contained within the **DCESecurityMechanismInfo** structure, the **target_requires** field may conform to the DCE semantics (i.e., no more than one option selected). If other security mechanisms are sharing the **TAG_ASSOCIATION_OPTIONS** component, and perhaps using different rules for interpreting the **target_requires** field, then the **target_requires** field may have several options selected. The "DCE Association Options Reduction" algorithm, described in "Secure DCE-CIOP Operational Semantics" on page 15-280, handles both cases and is used to select the appropriate DCE-secure invocation services given a set of required association options.

The **EstablishTrustInTarget** option in the **target_requires** field is meaningless, and is therefore ignored.

15.15.2.3 TAG_SEC_NAME

The security name component contains the DCE principal name of the target. Generally, this is a global principal name that includes the name of the cell in which the target principal's account resides. If a cell-relative principal name (i.e., the cell prefix does not appear) is specified, the local cell is assumed. Cell-relative principal names are only appropriate for use in IORs that are consumed by clients in the same cell in which the target resides. When an IOR containing a cell-relative principal name in the **TAG_SEC_NAME** component crosses a cell boundary, the cell-relative principal name should be replaced with a global name.

The format of a "human-friendly" DCE principal name is described in section 1.13 of [6]. It is a string containing a concatenated cell name and cell-relative principal name that looks like:

/.../cell-name/cell-relative-principal-name

For example, the principal with the cell-relative name "**printserver**" in the "**mis.prettybank.com**" cell has the global principal name:

/.../mis.prettybank.com/printserver

The **component_data** member of the **TAG_SEC_NAME** component is set to the string value of the DCE principal name. The string is represented directly in the sequence of octets, including the terminating **NULL**.

If the **TAG_SEC_NAME** component is not present, then a value of **NULL** is assumed, indicating that the client will depend on the DCE authenticated RPC runtime to retrieve the DCE principal name of the target, identified in the IOR by the DCE-CIOP string binding and binding name components. This case indicates that the client is not interested in authentication of the target identity.

15.15.3 DCE RPC Security Services

This section provides details about the protection provided by DCE Authenticated RPC authorization services, protection levels, and authentication services. See the **rpc_binding_set_auth_info()** man page in [9] for more information about using these protection parameters to secure an association between a client and target.

DCE RPC Authorization Services

This section describes the DCE authorization service indicated by the **authorization_service** member of the **DCESecurityMechanismInfo** structure in the **component_data** field of the **TAG_DCE_SEC_MECH** component.

DCEAuthorizationName indicates that the target performs authorization based on the client security name. The DCE RPC authorization service **DCEAuthorizationName** asserts the principal name (without cryptographic protection if the association option **NoProtection** is chosen, or with cryptographic protection otherwise).

DCEAuthorizationDCE indicates that the target performs authorization using the client's Privilege Attribute Certificate (for OSF DCE 1.0.3 or previous versions), or the client's Extended Privilege Attribute Certificate (for DCE 1.1). The authorization service **DCEAuthorizationDCE** asserts the principal name and appropriate authorization data (without cryptographic protection if the association option **NoProtection** is chosen, or with cryptographic protection otherwise).

DCEAuthorizationNone indicates that the target performs no authorization based on privilege information carried by the RPC runtime. This is valid only if the association option **NoProtection** is chosen.

The authorization_service identifiers defined here for Secure DCE-CIOP correspond to DCE RPC authorization service identifiers and are defined to have identical values. The relationship between these identifiers is shown in Table 15-24.

Table 15-24 Relation between DCE-CIOP and DCE RPC Authorization Service Identifiers

Secure DCE-CIOP authorization_service	DCE RPC Authorization Service	Shared Value
DCEAuthorizationNone	rpc_c_authz_none	0
DCEAuthorizationName	rpc_c_authz_name	1
DCEAuthorizationDCE	rpc_c_authz_dce	2

DCE RPC Protection Levels

The meanings of the DCE RPC protection levels referenced in Table 8-4 are described below. For the purposes of evaluating the protection levels, it is interesting to remember that a single DCE-CIOP message is transferred over the wire in the body of one or more DCE RPC PDUs.

rpc_c_protect_level_none indicates that no authentication or message protection is to be performed, regardless of the authentication service chosen. Depending on target policy, the client may be granted access as an unauthenticated principal.

rpc_c_protect_level_connect indicates that the client and server identities are exchanged and cryptographically verified at the time the binding is set up between them. Strong mutual authentication and replay detection *for the binding set-up only* is provided. There are no protection services per DCE RPC PDU.

rpc_c_protect_level_pkt indicates that the **rpc_c_protect_level_connect** services are provided plus detection of misordering or replay of DCE RPC PDUs. There is no protection against PDU modification.

rpc_c_protect_level_pkt_integrity offers the **rpc_c_protect_level_pkt** services plus detection of DCE RPC PDU modification.

rpc_c_protect_level_pkt_privacy offers the **rpc_c_protect_level_pkt_integrity** services plus privacy of RPC arguments, which means the DCE-CIOP message in its entirety is privacy-protected.

rpc_c_protect_level_default indicates the default protection level, as defined by the DCE implementation or by a site administrator (should be one of the above defined values).

DCE RPC Authentication Services

The meanings of the DCE RPC authentication services referenced in Table 15-24 are described below.

rpc_c_authn_none indicates no authentication. If this is selected, then no authorization, DCEAuthorizationNone, must be chosen as well.

rpc_c_authn_dce_secret indicates the DCE shared-secret key authentication service.

15.15.3.1 *Secure DCE-CIOP Operational Semantics*

This section describes how the DCE-CIOP transport layer should provide security for invocation and locate requests.

During a request invocation, if the IOR components indicate support for the DCE-CIOP transport and the **TAG_DCE_SEC_MECH** component is present, then a Secure DCE-CIOP request can be made.

Deriving DCE Security Parameters from Association Options

The client-side secure invocation policy and the target-side policy expressed in the **TAG_ASSOCIATION_OPTIONS** component are used to derive the actual options using the method described in "Determining Association Options" on page 15-182. These options are then reduced to a single **required_option** using the algorithm described in "The DCE Association Options Reduction Algorithm" on page 15-280 below. The resultant **required_option** is used to select a DCE RPC protection level and authentication service using Table 15-23 on page 15-276. The derived protection level and authentication service are used to secure the association via the **rpc_binding_set_auth_info()** call (see "Securing the Binding Handle to the Target" on page 15-281).

The DCE Association Options Reduction Algorithm

The "DCE Association Options Reduction" algorithm is used to select a single association option, **required_option,** given the value required by client and target derived as described in "Determining Association Options" on page 15-182. The resultant **required_option** indicates, via Table 15-23 on page 15-276, the DCE protection level and authentication service to use for invocations.

The association option names used in the following algorithm refer to options in the negotiated-required options set.

The "DCE Association Options Reduction" algorithm is expressed as:

If Confidentiality is set, then required_option = Confidentiality;
else if Integrity is set, then required_option = Integrity;
else if DetectReplay is set, OR
if DetectMisordering is set,
then required_option = DetectReplay;
(alternatively, the same results are obtained with:
then required_option = DetectMisordering;)
else if EstablishTrustInClient is set,
then required_option = EstablishTrustInClient;
else required_option = NoProtection.

Behavior When TAG_ASSOCIATION_OPTIONS Not Present

As described earlier, if the **TAG_ASSOCIATION_OPTIONS** component is not present, then the target is assumed to support and require **rpc_c_protect_level_default** and **rpc_c_authn_dce_secret**. Since these protection parameters are not expressed as association options, the usual method of deriving a single **required_option** by combining client and target policy (see "Determining Association Options" on page 15-182 and "The DCE Association Options Reduction Algorithm" on page 15-280"above) cannot be used. Instead, use the following alternative method to derive the required DCE RPC protection level and authentication service:

- Translate the client-side secure invocation policy from a set of client-supported association options to a single **client_supported_option** and from a set of client-required association options to a single **client_required_option**, using in each case the algorithm described in "The DCE Association Options Reduction Algorithm" on page 15-280.
- Using Table 15-23 "Association Option Mapping to DCE Security" translate the **client_supported_option** and **client_required_option** to corresponding "supported" and "required" DCE RPC protection level/authentication service pairs.
- If the target principal is a member of the local cell, determine the target-required protection level implied by **rpc_c_protect_level_default** by calling **rpc_mgmt_inq_dflt_protect_level()** passing **rpc_c_authn_dce_secret** as the **authn_svc** parameter. If the target principal is not a member of the local cell or if it's difficult to determine, then assume a target-required protection level of **rpc_c_protect_level_pkt_integrity**.
- If the client supports **rpc_c_authn_dce_secret**, then choose the strongest protection level that both the client and target support and that does not exceed the strongest protection level required by either the client or target. If the client does not support **rpc_c_authn_dce_secret**, then choose **rpc_c_authn_none** and **rpc_c_protect_level_none**. Use the protection level and authentication service thus derived to secure the association between this client and target.

Securing the Binding Handle to the Target

The DCE-CIOP protocol engine acquires an **rpc_binding_handle** to the target using its normal procedure. The **DCE_CIOP** sets authentication and authorization information on that binding handle with the **rpc_binding_set_auth_info()** call using data from the IOR profile security components in the following way:

- The target security name string from the **TAG_SEC_NAME** component (or NUL, if the component is not present) is passed to **rpc_binding_set_auth_info()** via the **server_princ_name** parameter.
- If the **TAG_ASSOCIATION_OPTIONS** component is present in the IOR, see "Deriving DCE Security Parameters from Association Options" on page 15-280 above to select a DCE RPC protection level and authentication service for this invocation.

  If the **TAG_ASSOCIATION_OPTIONS** component is not present in the IOR, see "Behavior When TAG_ASSOCIATION_OPTIONS Not Present" on page 15-281 above to select a DCE RPC protection level and authentication service for this invocation.

  The selected protection level is passed to **rpc_binding_set_auth_info()** via the **protect_level** parameter. The selected authentication service is passed via the **authn_svc** parameter to **rpc_binding_set_auth_info()**.
- The **auth_identity** parameter is set to **NULL** to use the DCE default login context.
- The authorization service identifier from the **authorization_service** field of the **DCESecurityMechanismInfo component_data** is mapped to the corresponding DCE RPC authorization service identifier (using Table 15-24 on page 15-279) which is then passed via the **authz_svc** parameter.

After a successful call to **rpc_binding_set_auth_info()**, the authenticated binding handle will be used by the DCE-CIOP protocol engine to make secure requests.

Appendix A Consolidated OMG IDL

A.9 Introduction

The OMG IDL for CORBA security is split into modules as follows:

- A module containing the common data types used by all security modules.
- A module for application interfaces for each Security Functionality Levels 1 and 2.
- A module for Security Level 2 security policy administration.
- A module for non-repudiation, including the non-repudiation policy administration interface.
- A module for the Replaceable Security Service, as described in Section 15.7, “Implementor’s Security Interfaces,” on page 15-156.
- A module for elements of the SECure Inter Orb Protocol (SECIOP)l.
- A module for elements of the SSL Protocol.
- A module for elements related to Security that are added to the DCE_CIOPSecurity module.

A.10 General Security Data Module

This subsection defines the OMG IDL for security data types common to the other security modules, which is the module **Security**. The **Security** module depends on the **TimeBase** module and the **CORBA** module.

Note – If your IDL compiler does not yet support the “Escaped Identifiers” (which provides for the addition of new keywords to IDL, compile this module with the preprocessor definition “NO_ESCPAED_IDENTIFIERS”. With many compilers this would be done with a qualifier on the command line, something like -
-DNO_ESCAPED_IDENTIFIERS.

```
#ifndef_SECURITY_IDL
#define _SECURITY_IDL

#include <orb.idl>
#include <TimeBase.idl>

#pragma prefix "omg.org"

module Security {

    typedef string                  SecurityName;
    typedef sequence <octet>        Opaque;

    // Constant declarations for Security Service Options
```

```
const CORBA::ServiceOption SecurityLevel1 = 1;
const CORBA::ServiceOption SecurityLevel2 = 2;
const CORBA::ServiceOption NonRepudiation = 3;
const CORBA::ServiceOption SecurityORBServiceReady = 4;
const CORBA::ServiceOption SecurityServiceReady = 5;
const CORBA::ServiceOption ReplaceORBServices = 6;
const CORBA::ServiceOption ReplaceSecurityServices = 7;
const CORBA::ServiceOption StandardSecureInteroperability = 8;
const CORBA::ServiceOption DCESecureInteroperability = 9;

// Service options for Common Secure Interoperability

const CORBA::ServiceOption CommonInteroperabilityLevel0 = 10;
const CORBA::ServiceOption CommonInteroperabilityLevel1 = 11;
const CORBA::ServiceOption CommonInteroperabilityLevel2 = 12;

// Security mech types supported for secure association
const CORBA::ServiceDetailType SecurityMechanismType = 1;

// privilege types supported in standard access policy
const CORBA::ServiceDetailType SecurityAttribute = 2;

// extensible families for standard data types

struct ExtensibleFamily {
    unsigned short      family_definer;
    unsigned short      family;
};

// security association mechanism type

typedef string              MechanismType;
struct                      SecurityMechandName {
    MechanismType       mech_type;
    SecurityName        security_name;
};

typedef sequence<MechanismType> MechanismTypeList;
typedef sequence<SecurityMechandName> SecurityMechandNameList;

// security attributes

typedef unsigned long       SecurityAttributeType;

// other attributes; family = 0

const SecurityAttributeType         AuditId = 1;
const SecurityAttributeType         AccountingId = 2;
const SecurityAttributeType         NonRepudiationId = 3;
```

```
    // privilege attributes; family = 1
#ifdef NO_ESCAPED_IDENTIFIERS
    const SecurityAttributeType         Public = 1;
#else
    const SecurityAttributeType         _Public = 1;
#endif
    const SecurityAttributeType         AccessId = 2;
    const SecurityAttributeType         PrimaryGroupId = 3;
    const SecurityAttributeType         GroupId = 4;
    const SecurityAttributeType         Role = 5;
    const SecurityAttributeType         AttributeSet = 6;
    const SecurityAttributeType         Clearance = 7;
    const SecurityAttributeType         Capability = 8;

    struct AttributeType {
        ExtensibleFamily            attribute_family;
        SecurityAttributeType       attribute_type;
    };

    typedef sequence<AttributeType>       AttributeTypeList;

    struct SecAttribute {
        AttributeType           attribute_type;
        Opaque                  defining_authority;
        Opaque                  value;
        // the value of this attribute can be
        // interpreted only with knowledge of type
    };

    typedef sequence <SecAttribute> AttributeList;


    // Authentication return status

    enum AuthenticationStatus {
        SecAuthSuccess,
        SecAuthFailure,
        SecAuthContinue,
        SecAuthExpired
    };

    // Association return status
    enum AssociationStatus {
        SecAssocSuccess,
        SecAssocFailure,
        SecAssocContinue
    };

    // Authentication method
    typedef unsigned long AuthenticationMethod;
```

```
// Credential types which can be set as Current default
enum CredentialType {
    SecInvocationCredentials,
    SecNRCredentials
};

// Declarations related to Rights
struct Right {
    ExtensibleFamily        rights_family;
    string                  right;
};

typedef sequence <Right> RightsList;

enum RightsCombinator {
    SecAllRights,
    SecAnyRight
};

// Delegation related
enum DelegationState {
    SecInitiator,
    SecDelegate
};

// pick up from TimeBase
typedef TimeBase::UtcT          UtcT;
typedef TimeBase::IntervalT     IntervalT;
typedef TimeBase::TimeT         TimeT;

// Security features available on credentials.
enum SecurityFeature {
    SecNoDelegation,
    SecSimpleDelegation,
    SecCompositeDelegation,
    SecNoProtection,
    SecIntegrity,
    SecConfidentiality,
    SecIntegrityAndConfidentiality,
    SecDetectReplay,
    SecDetectMisordering,
    SecEstablishTrustInTarget
};

// Security feature-value
struct SecurityFeatureValue {
    SecurityFeature         feature;
    boolean                 value;
};

typedef sequence <SecurityFeatureValue> SecurityFeatureValueList;
```

```
// Quality of protection which can be specified
// for an object reference and used to protect messages
enum QOP {
    SecQOPNoProtection,
    SecQOPIntegrity,
    SecQOPConfidentiality,
    SecQOPIntegrityAndConfidentiality
};

// Association options which can be administered
// on secure invocation policy and used to
// initialize security context

typedef unsigned short        AssociationOptions;

const AssociationOptions NoProtection = 1;
const AssociationOptions Integrity = 2;
const AssociationOptions Confidentiality = 4;
const AssociationOptions DetectReplay = 8;
const AssociationOptions DetectMisordering = 16;
const AssociationOptions EstablishTrustInTarget = 32;
const AssociationOptions EstablishTrustInClient = 64;

// Flag to indicate whether association options being
// administered are the "required" or "supported" set

enum RequiresSupports {
    SecRequires,
    SecSupports
};

// Direction of communication for which
// secure invocation policy applies

enum CommunicationDirection {
    SecDirectionBoth,
    SecDirectionRequest,
    SecDirectionReply
};

// AssociationOptions-Direction pair

struct OptionsDirectionPair {
    AssociationOptions    options;
    CommunicationDirectiondirection;
};

typedef sequence <OptionsDirectionPair> OptionsDirectionPairList;
```

```
// Delegation mode which can be administered

enum DelegationMode {
    SecDelModeNoDelegation,          // i.e. use own credentials
    SecDelModeSimpleDelegation,      // delegate received credentials
    SecDelModeCompositeDelegation  // delegate both;
    };

// Association options supported by a given mech type
struct MechandOptions {
    MechanismType          mechanism_type;
    AssociationOptions     options_supported;
};

typedef sequence <MechandOptions>       MechandOptionsList;


// Audit
typedef unsigned long                  AuditChannelId;
typedef unsigned short                 EventType;

const EventType        AuditAll = 0;
const EventType        AuditPrincipalAuth = 1;
const EventType        AuditSessionAuth = 2;
const EventType        AuditAuthorization = 3;
const EventType        AuditInvocation = 4;
const EventType        AuditSecEnvChange = 5;
const EventType        AuditPolicyChange = 6;
const EventType        AuditObjectjCreation = 7;
const EventType        AuditObjectDestruction = 8;
const EventType        AuditNonRepudiation = 9;

struct AuditEventType {
    ExtensibleFamily       event_family;
    EventType              event_type;
};

typedef sequence <AuditEventType>       AuditEventTypeList;

typedef unsigned long                  SelectorType;

const SelectorType     InterfaceRef = 1;
const SelectorType     ObjectRef = 2;
const SelectorType     Operation = 3;
const SelectorType     Initiator = 4;
const SelectorType     SuccessFailure = 5;
const SelectorType     Time = 6;

// values defined for audit_needed and audit_write are:
// InterfaceRef: object reference
// ObjectRef: object reference
```

```
    // Operation: op_name
    // Initiator: Credentials
    // SuccessFailure: boolean
    // Time: utc time on audit_write; time picked up from
    // environment in audit_needed if required

    struct SelectorValue {
        SelectorType        selector;
        any                 value;
    };
    typedef sequence <SelectorValue>    SelectorValueList;

    // Constant declaration for valid Security Policy Types
    // General administrative policies
    const CORBA::PolicyType SecClientInvocationAccess = 1;
    const CORBA::PolicyType SecTargetInvocationAccess = 2;
    const CORBA::PolicyType SecApplicationAccess = 3;
    const CORBA::PolicyType SecClientInvocationAudit = 4;
    const CORBA::PolicyType SecTargetInvocationAudit = 5;
    const CORBA::PolicyType SecApplicationAudit = 6;
    const CORBA::PolicyType SecDelegation = 7;
    const CORBA::PolicyType SecClientSecureInvocation = 8;
    const CORBA::PolicyType SecTargetSecureInvocation = 9;
    const CORBA::PolicyType SecNonRepudiation = 10;

    // Policies used to control attributes of a binding to a target
    const CORBA::PolicyType SecMechanismsPolicy = 12;
    const CORBA::PolicyType SecCredentialsPolicy = 13;
    const CORBA::PolicyType SecFeaturesPolicy = 14;
    const CORBA::PolicyType SecQOPPolicy = 15;

};
#endif /* _SECURITY_IDL */
```

A.11 Application Interfaces - Level 1

This subsection defines those interfaces available to application objects using only Security Functionality Level 1, and consists of a single module, **SecurityLevel1**. This module depends on the **CORBA** module, and on the **Security** module.

```
#ifndef _SECURITY_LEVEL_1_IDL
#define _SECURITY_LEVEL_1_IDL

#include <Security.idl>

#pragma prefix "omg.org"

module SecurityLevel1 {
    interface Current : CORBA::Current {// Locality-Constrained
```

```
            // thread-specific operations

            Security::AttributeList get_attributes (
                in   Security::AttributeTypeList     attributes
            );
        };
};
#endif /* _SECURITY_LEVEL_1_IDL */
```

A.12 Application Interfaces - Level 2

This subsection defines the interfaces available to applications using Security Functionality Level 2, all of which are declared in the **SecurityLevel2** module. This module depends on the **CORBA**, **SecurityLevel1,** and **Security** modules. The interfaces are described in Section 15.5, "Application Developer's Interfaces," on page 15-88.

```
#ifndef _SECURITY_LEVEL_2_IDL
#define _SECURITY_LEVEL_2_IDL

#include <SecurityLevel1.idl>

#pragma prefix "omg.org"

module SecurityLevel2 {

    // Forward declaration of interfaces
    interface PrincipalAuthenticator;
    interface Credentials;
    interface Current

    // Interface PrincipalAuthenticator
        interface PrincipalAuthenticator {          // Locality-Constrained

            Security::AuthenticationStatus authenticate (
                in      Security::AuthenticationMethod   method,
                in      Security::SecurityName           security_name,
                in      Security::Opaque                 auth_data,
                in      Security::AttributeList          privileges,
                out     Credentials                      creds,
                out     Security::Opaque                 continuation_data,
                out     Security::Opaque                 auth_specific_data
            );

            Security::AuthenticationStatus continue_authentication (
                in      Security::Opaque                 response_data,
                in      Credentials                      creds,
                out     Security::Opaque                 continuation_data,
                out     Security::Opaque                 auth_specific_data
            );
```

```
};

// Interface Credentials
interface Credentials {         // Locality-Constrained

    Credentials copy ();

    void destroy();

    void set_security_features (
        in      Security::CommunicationDirection    direction,
        in      Security::SecurityFeatureValueList  security_features
    );

    Security::SecurityFeatureValueList get_security_features(
        in      Security::CommunicationDirection    direction
    );

    boolean set_privileges (
        in      boolean                         force_commit,
        in      Security::AttributeList         requested_privileges,
        out     Security::AttributeList         actual_privileges
    );

    Security::AttributeList get_attributes (
        in      Security::AttributeTypeList     attributes
    );

    boolean is_valid (
        out     Security::UtcT                  expiry_time
    );

    boolean refresh();
};

typedef sequence <Credentials>          CredentialsList;


// RequiredRights Interface

interface RequiredRights{
    void get_required_rights(
        in      Object                          obj,
        in      CORBA::Identifier               operation_name,
        in      CORBA::RepositoryId             interface_name,
        out     Security::RightsList            rights,
        out     Security::RightsCombinator      rights_combinator
    );

    void set_required_rights(
        in      string                          operation_name,
```

```
            in      CORBA::RepositoryId          interface_name,
            in      Security::RightsList         rights,
            in      Security::RightsCombinator   rights_combinator
        );
};

// interface audit channel
interface AuditChannel {                   // Locality-Constrained

        void audit_write (
            in      Security::AuditEventType     event_type,
            in      CredentialsList              creds,
            in      Security::UtcT               time,
            in      Security::SelectorValueList  descriptors,
            in      Security::Opaque             event_specific_data
        );

        readonly attribute Security::AuditChannelI    audit_channel_id;
};

// interface for Audit Decision
interface AuditDecision {                    // Locality-Constrained

        boolean audit_needed (
            in      Security::AuditEventType             event_type,
            in      Security::SelectorValueList          value_list
        );

        readonly attribute AuditChannel audit_channel;
};

interface AccessDecision {                   // Locality-Constrained

        boolean access_allowed (
            in      SecurityLevel2::CredentialsList   cred_list,
            in      Object                            target,
            in      CORBA::Identifier                 operation_name,
            in      CORBA::Identifier                 target_interface_name
        );
};

// Policy interfaces to control bindings

interface QOPPolicy : CORBA::Policy {        // Locality-Constrained
        readonly attribute Security::QOP                 qop;
};

interface MechanismPolicy : CORBA::Policy {// Locality-Constrained
        readonly attribute Security::MechanismTypeList  mechanisms;
};
```

```
interface SecurityFeaturesPolicy : CORBA::Policy {
                                            // Locality-Constrained
    readonly attribute Security::SecurityFeatureValueList
                                                features;
};

interface InvocationCredentialsPolicy : CORBA::Policy {
                                            // Locality-Constrained
    readonly attribute CredentialsList          creds;
};

enum DelegationMode {Delegate, NoDelegate};

// Interface Current derived from SecurityLevel1::Current
// providing additional operations on Current at this
// security level. This is implemented by the ORB

interface Current : SecurityLevel1::Current {  // Locality-Constrained

// Thread-specific operations

    readonly attribute CredentialsList received_credentials;
    readonly attribute CredentialsList own_credentials;
    readonly attribute Security::SecurityFeatureValueList
                                received_security_features;

    void set_credentials (
        in      Security::CredentialType    cred_type,
        in      CredentialsList             creds,
        in      DelegationMode              del
    );

    CredentialsList get_credentials (
        in      Security::CredentialType    cred_type
    );

    CORBA::Policy get_policy (
        in      CORBA::PolicyType           policy_type
    );

// Process/Capsule/ORB Instance-specific operations

    readonly attribute RequiredRights       required_rights_object;
    readonly attribute PrincipalAuthenticator
                                            principal_authenticator;
    readonly attribute AccessDecision       access_decision;
    readonly attribute AuditDecision        audit_decision;

    // security names for given target
    Security::SecurityMechandNameList get_security_names (
        in      Object                          obj_ref
```

```
            );

            // Factory operations for local policies controlling bindings
            QOPPolicy create_qop_policy(
                in      Security::QOP                   qop
            );

            MechanismPolicy create_mechanism_policy(
                in      Security::MechanismTypeList     mechanisms
            );

            InvocationCredentialsPolicy create_invoc_creds_policy(
                in      CredentialsList                 creds
            );
        }
};

#endif /* _SECURITY_LEVEL_2_IDL */
```

A.13 Security Administration Interfaces

This section covers interfaces concerned with querying and modifying security policies, and comprises the module **SecurityAdmin**. The **SecurityAdmin** module depends on **CORBA**, **Security**, and **SecurityLevel2** modules. The interfaces are described in “Administrator’s Interfaces” on page 15-130. There are related interfaces for finding domain managers and policies. They are to be found in the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*.

```
#ifndef _SECURITY_ADMIN_IDL
#define _SECURITY_ADMIN_IDL

#include <SecurityLevel2.idl>

#pragma prefix "omg.org"


module SecurityAdmin {

    // interface AccessPolicy
    interface AccessPolicy : CORBA::Policy {

        Security::RightsList get_effective_rights (
            in      Security::AttributeList         attrib_list,
            in      Security::ExtensibleFamily      rights_family
        );
};

    // interface DomainAccessPolicy
    interface DomainAccessPolicy : AccessPolicy {
```

```
        void grant_rights(
            in      Security::SecAttribute          priv_attr,
            in      Security::DelegationState       del_state,
            in      Security::ExtensibleFamily      rights_family,
            in      Security::RightsList            rights
        );

        void revoke_rights(
            in      Security::SecAttribute          priv_attr,
            in      Security::DelegationState       del_state,
            in      Security::ExtensibleFamily      rights_family,
            in      Security::RightsList            rights
        );

            void replace_rights (
            in      Security::SecAttribute          priv_attr,
            in      Security::DelegationState       del_state,
            in      Security::ExtensibleFamily      rights_family,
            in      Security::RightsList            rights
        );

        Security::RightsList get_rights (
            in      Security::SecAttribute          priv_attr,
            in      Security::DelegationState       del_state,
            in      Security::ExtensibleFamily      rights_family
        );
    };

    // interface AuditPolicy
    interface AuditPolicy : CORBA::Policy {

        void set_audit_selectors (
            in      CORBA::InterfaceDef             object_type,
            in      Security::AuditEventTypeList    events,
            in      Security::SelectorValueList     selectors
        );

        void clear_audit_selectors (
            in      CORBA::InterfaceDef             object_type,
            in      Security::AuditEventTypeList    events
        );

        void replace_audit_selectors (
            in      CORBA::InterfaceDef             object_type,
            in      Security::AuditEventTypeList    events,
            in      Security::SelectorValueList     selectors
        );

        Security::SelectorValueList get_audit_selectors (
            in      CORBA::InterfaceDef             object_type,
            in      Security::AuditEventTypeList    events
```

```
        );

        void set_audit_channel (
            in      Security::AuditChannelId            audit_channel_id
        );
    };

    // interface SecureInvocationPolicy
    interface SecureInvocationPolicy : CORBA::Policy {

        void set_association_options(
            in      CORBA::InterfaceDef                 object_type,
            in      Security::RequiresSupports          requires_supports,
            in      Security::CommunicationDirectiondirection,
            in      Security::AssociationOptions        options
        );

        Security::AssociationOptions get_association_options(
            in      CORBA::InterfaceDef                 object_type,
            in      Security::RequiresSupports          requires_supports,
            in      Security::CommunicationDirectiondirection
        );
    };

    // interface DelegationPolicy
    interface DelegationPolicy : CORBA::Policy {

        void set_delegation_mode(
            in      CORBA::InterfaceDef                 object_type,
            in      Security::DelegationMode            mode
        );

        Security::DelegationMode get_delegation_mode(
            in      CORBA::InterfaceDef                 object_type
        );
    };
};

#endif /* _SECURITY_ADMIN_IDL */
```

A.14 Interfaces for Non-repudiation

This subsection defines the optional application interface for non-repudiation. This module depends on **SecurityLevel2** and **CORBA** modules. The interfaces are described in “Non-repudiation” on page 15-121.

```
#ifndef _NR_SERVICE_IDL
#define _NR_SERVICE_IDL

#include <SecurityLevel2.idl>
```

```
#pragma prefix "omg.org"

module NRService {
    typedef Security::MechanismType        NRMech;
    typedef Security::ExtensibleFamily     NRPolicyId;

    enum EvidenceType {
        SecProofofCreation,
        SecProofofReceipt,
        SecProofofApproval,
        SecProofofRetrieval,
        SecProofofOrigin,
        SecProofofDelivery,
        SecNoEvidence    // used when request-only token desired
    };

    enum NRVerificationResult {
        SecNRInvalid,
        SecNRValid,
        SecNRConditionallyValid
    };

    // the following are used for evidence validity duration
    typedef unsigned long    DurationInMinutes;

    const DurationInMinutes DurationHour   = 60;
    const DurationInMinutes DurationDay    = 1440;
    const DurationInMinutes DurationWeek   = 10080;
    const DurationInMinutes DurationMonth = 43200;// 30 days;
    const DurationInMinutes DurationYear   = 525600;//365 days;

    typedef long TimeOffsetInMinutes;

    struct NRPolicyFeatures {
        NRPolicyId         policy_id;
        unsigned long      policy_version;
        NRMech             mechanism;
    };

    typedef sequence <NRPolicyFeatures> NRPolicyFeaturesList;

    // features used when generating requests
    struct RequestFeatures {
        NRPolicyFeatures       requested_policy;
        EvidenceType           requested_evidence;
        string                 requested_evidence_generators;
        string                 requested_evidence_recipients;
        boolean                include_this_token_in_evidence;
    };
```

```
struct EvidenceDescriptor {
    EvidenceType            evidence_type;
    DurationInMinutes       evidence_validity_duration;
    boolean                 must_use_trusted_time;
};

typedef sequence <EvidenceDescriptor> EvidenceDescriptorList;

struct AuthorityDescriptor {
    string                  authority_name;
    string                  authority_role;
    TimeOffsetInMinutes     last_revocation_check_offset;
                // may be >0 or <0; add this to evid. gen. time to
                // get latest time at which mech. will check to see
                // if this authority's key has been revoked.
};

typedef sequence <AuthorityDescriptor> AuthorityDescriptorList;

struct MechanismDescriptor {
    NRMech                  mech_type;
    AuthorityDescriptorList authority_list;
    TimeOffsetInMinutes     max_time_skew;
            // max permissible difference between evid. gen. time
            // and time of time service countersignature
            // ignored if trusted time not reqd.
};

typedef sequence <MechanismDescriptor> MechanismDescriptorList;

interface NRCredentials : SecurityLevel2::Credentials{

    boolean set_NR_features (
        in      NRPolicyFeaturesList    requested_features,
        out     NRPolicyFeaturesList    actual_features
    );

    NRPolicyFeaturesList get_NR_features ();

    void generate_token (
        in      Security::Opaque        input_buffer,
        in      EvidenceType            generate_evidence_type,
        in      boolean                 include_data_in_token,
        in      boolean                 generate_request,
        in      RequestFeatures         request_features,
        in      boolean                 input_buffer_complete,
        out     Security::Opaque        nr_token,
        out     Security::Opaque        evidence_check
    );

    NRVerificationResult verify_evidence (
```

```
        in      Security::Opaque        input_token_buffer,
        in      Security::Opaque        evidence_check,
        in      boolean                 form_complete_evidence,
        in      boolean                 token_buffer_complete,
        out     Security::Opaque        output_token,
        out     Security::Opaque        data_included_in_token,
        out     boolean                 evidence_is_complete,
        out     boolean                 trusted_time_used,
        out     Security::TimeT         complete_evidence_before,
        out     Security::TimeT         complete_evidence_after
);

    void get_token_details (
        in      Security::Opaque        token_buffer,
        in      boolean                 token_buffer_complete,
        out     string                  token_generator_name,
        out     NRPolicyFeatures        policy_features,
        out     EvidenceType            evidence_type,
        out     Security::UtcT          evidence_generation_time,
        out     Security::UtcT          evidence_valid_start_time,
        out     DurationInMinutes       evidence_validity_duration,
        out     boolean                 data_included_in_token,
        out     boolean                 request_included_in_token,
        out     RequestFeatures         request_features
);

    boolean form_complete_evidence (
        in      Security::Opaque        input_token,
        out     Security::Opaque        output_token,
        out     boolean                 trusted_time_used,
        out     Security::TimeT         complete_evidence_before,
        out     Security::TimeT         complete_evidence_after
    );
};

interface NRPolicy : CORBA::Policy{

    void get_NR_policy_info   (
        out     Security::ExtensibleFamilyNR_policy_id,
        out     unsigned long           policy_version,
        out     Security::TimeT         policy_effective_time,
        out     Security::TimeT         policy_expiry_time,
        out     EvidenceDescriptorList  supported_evidence_types,
        out     MechanismDescriptorList supported_mechanisms
    );

    boolean set_NR_policy_info (
        in      MechanismDescriptorList     requested_mechanisms,
        out     MechanismDescriptorList     actual_mechanisms
    );
};
```

```
};
#endif /* _NR_SERVICE_IDL */
```

A.15 Security Replaceable Service Interfaces

This section defines the IDL interfaces to the Security objects, which should be replaced if there is a requirement to replace the Security services used for security associations (i.e., the **Vault** and **Security Context**). The IDL provided here is for those interfaces that have not already been covered by the **SecurityLevel2** module. This section comprises the module **SecurityReplaceable**. This module depends on the **CORBA**, **Security**, and **SecurityLevel2** modules. The interfaces are described in Section 15.7, “Implementor’s Security Interfaces,” on page 15-156.

```
#ifndef _SECURITY_REPLACEABLE_IDL
#define _SECURITY_REPLACEABLE_IDL

#include <SecurityLevel2.idl>

#pragma prefix "omg.org"


module SecurityReplaceable {

    interface SecurityContext;

    interface Vault {                       // Locality-Constrained

        Security::AssociationStatus init_security_context (
            in      SecurityLevel2::CredentialsList    creds_list,
            in      Security::SecurityName             target_security_name,
            in      Object                             target,
            in      Security::DelegationMode           delegation_mode,
            in      Security::OptionsDirectionPairListassociation_options,
            in      Security::MechanismType            mechanism,
            in      Security::Opaque                   mech_data, //from IOR
            in      Security::Opaque                   chan_binding,
            out     Security::Opaque                   security_token,
        );

        Security::AssociationStatus accept_security_context (
            in      SecurityLevel2::CredentialsList    creds_list,
            in      Security::Opaque                   chan_bindings,
            in      Security::Opaque                   in_token,
            out     Security::Opaque                   out_token,
            out     SecurityContext                    security_context
        );

        Security::MechandOptionsList get_supported_mechs ();
    };
```

```
    interface SecurityContext {          // Locality-Constrained

        readonly attribute SecurityLevel2::CredentialsList
                                            received_credentials;
        readonly attribute Security::SecurityFeatureValueList
                                            security_features;

        Security::AssociationStatus continue_security_context (
            in      Security::Opaque            in_token,
            out     Security::Opaque            out_token
        );

        void protect_message (
            in      Security::Opaque            message,
            in      Security::QOP               qop,
            out     Security::Opaque            text_buffer,
            out     Security::Opaque            token
        );

        boolean reclaim_message (
            in      Security::Opaque            text_buffer,
            in      Security::Opaque            token,
            out     Security::QOP               qop,
            out     Security::Opaque            message
        );

        boolean is_valid (
            out     Security::UtcT                  expiry_time
        );

        boolean refresh ();
    };

};

#endif /* _SECURITY_REPLACEABLE_IDL */
```

A.16 Secure Inter-ORB Protocol (SECIOP)

The **SECIOP** module holds structure declarations related to the layout of message fields in the Secure Inter-ORB protocol. This module depends on the **IOP** and **Security** modules.

Note – If your IDL compiler does not yet support the “long long” data type, compile this module with the preprocessor definition “NOLONGLONG”. With many compilers this would be done with a qualifier on the command line, something like -
-DNOLONGLONG.

```
#ifndef _SECIOP_IDL
#define _SECIOP_IDL

#include <IOP.idl>
#include <Security.idl>

#pragma prefix "omg.org"

module SECIOP {

    const IOP::ComponentId TAG_GENERIC_SEC_MECH = 22;

    const IOP::ComponentId TAG_ASSOCIATION_OPTIONS = 13;

    const IOP::ComponentId TAG_SEC_NAME = 14;

    struct TargetAssociationOptions{
            Security::AssociationOptions     target_supports;
            Security::AssociationOptions     target_requires;
    };

    struct GenericMechanismInfo {
            sequence <octet>                 security_mechanism_type;
            sequence <octet>                 mech_specific_data;
            sequence <IOP::TaggedComponent>      components;
    };

    enum MsgType {
        MTEstablishContext,
        MTCompleteEstablishContext,
        MTContinueEstablishContext,
        MTDiscardContext,
        MTMessageError,
        MTMessageInContext
    };

#ifdef NOLONGLONG
    struct ulonglong {
            unsigned long low;
            unsigned long high;
    };
    typedef ulonglong ContextId;
#else
    typedef unsigned long long ContextId;
#endif
    enum ContextIdDefn {
            CIDClient,
            CIDPeer,
            CIDSender
    };
```

```
struct EstablishContext {
        ContextId               client_context_id;
        sequence <octet>        initial_context_token;
};

struct CompleteEstablishContext {
        ContextId               client_context_id;
        boolean                 target_context_id_valid;
        ContextId               target_context_id;
        sequence <octet>        final_context_token;
};

struct ContinueEstablishContext {
        ContextId               client_context_id;
        sequence <octet>        continuation_context_token;
};

struct DiscardContext {
        ContextIdDefn           message_context_id_defn;
        ContextId               message_context_id;
        sequence <octet>        discard_context_token;
};

struct MessageError {
        ContextIdDefn           message_context_id_defn;
        ContextId               message_context_id;
        long                    major_status;
        long                    minor_status;
};

enum ContextTokenType {
        SecTokenTypeWrap,
        SecTokenTypeMIC
};

struct MessageInContext {
        ContextIdDefn           message_context_id_defn;
        ContextId               message_context_id;
        ContextTokenType        message_context_type;
        sequence <octet>        message_protection_token;
};

// message_protection_token is obtained by CDR encoding
// the following SequencingHeader followed by the octets of the
// frame data. SequencingHeader + Frame Data is called a
// SequencedDataFrame

struct SequencingHeader {
    octet                   control_state;
    unsigned long           direct_sequence_number;
    unsigned long           reverse_sequence_number;
```

```
    unsigned long           reverse_window;
};

typedef sequence <octet> SecurityName;
typedef unsigned short CryptographicProfile;
typedef sequence <CryptographicProfile> CryptographicProfileList;

// Cryptographic profiles for SPKM

const CryptographicProfile     MD5_RSA = 20;
const CryptographicProfile     MD5_DES_CBC = 21;
const CryptographicProfile     DES_CBC = 22;
const CryptographicProfile     MD5_DES_CBC_SOURCE  = 23;
const CryptographicProfile     DES_CBC_SOURCE  = 24;

// Security Mechanism SPKM_1

const IOP::ComponentId         TAG_SPKM_1_SEC_MECH = 15;

struct SPKM_1 {
    Security::AssociationOptions      target_supports;
    Security::AssociationOptions      target_requires;
    CryptographicProfileList          crypto_profile;
    SecurityName                      security_name;
};

// Security Mechanism SPKM_2

const IOP::ComponentId TAG_SPKM_2_SEC_MECH = 16;

struct SPKM_2 {
    Security::AssociationOptions      target_supports;
    Security::AssociationOptions      target_requires;
    CryptographicProfileList          crypto_profile;
    SecurityName                      security_name;
};

// Cryptographic profiles for GSS Kerberos Protocol

const CryptographicProfile          DES_CBC_DES_MAC = 10;
const CryptographicProfile          DES_CBC_MD5 = 11;
const CryptographicProfile          DES_MAC = 12;
const CryptographicProfile          MD5 = 13;

// Security Mechanism KerberosV5

const IOP::ComponentId TAG_KerberosV5_SEC_MECH = 17;

struct KerberosV5 {
    Security::AssociationOptions      target_supports;
    Security::AssociationOptions      target_requires;
```

```
        CryptographicProfileList        crypto_profile;
        SecurityName                    security_name;
    };

    // Cryptographic profiles for CSI-ECMA Protocol

    const CryptographicProfile      FullSecurity = 1;
    const CryptographicProfile      NoDataConfidentiality = 2;
    const CryptographicProfile      LowGradeConfidentiality = 3;
    const CryptographicProfile      AgreedDefault = 5;

    // Security Mechanism CSI_ECMA_Secret

    const IOP::ComponentId TAG_CSI_ECMA_Secret_SEC_MECH = 18;

    struct CSI_ECMA_Secret {
        Security::AssociationOptions    target_supports;
        Security::AssociationOptions    target_requires;
        CryptographicProfileList        crypto_profile;
        SecurityName                    security_name;
    };

    // Security Mechanism CSI_ECMA_Hybrid

    const IOP::ComponentId TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;

    struct CSI_ECMA_Hybrid {
        Security::AssociationOptions    target_supports;
        Security::AssociationOptions    target_requires;
        CryptographicProfileList        crypto_profile;
        SecurityName                    security_name;
    };

    // Security Mechanism CSI_ECMA_Public

    const IOP::ComponentId TAG_CSI_ECMA_Public_SEC_MECH = 21;

    struct CSI_ECMA_Public {
        Security::AssociationOptions    target_supports;
        Security::AssociationOptions    target_requires;
        CryptographicProfileList        crypto_profile;
        SecurityName                    security_name;
    };
};
#endif /* _SECIOP_IDL */
```

A.17 SSL

The **SSLIOP** module holds the structure and TAG definitions needed for using SSL as the secure transport under CORBA Security. This module depends on the **Security** and the **IOP** modules.

```
#ifndef _SSLIOP_IDL
#define _SSLIOP_IDL

#include <IOP.idl>
#include<Security.idl>

#pragma prefix "omg.org"

module SSLIOP {

    // Security mechanism SSL
    const IOP::ComponentId        TAG_SSL_SEC_TRANS = 20;

    struct SSL {
        Security::AssociationOptions      target_supports;
        Security::AssociationOptions      target_requires;
        unsigned short                    port;
    };
};
#endif /* _SSLIOP_IDL */
```

A.18 Secure DCE CIOP

The **DCE_CIOP_Security** module extension holds structures and TAG definitions needed for using DCE-CIOP Security. This module depends on **Security** and **IOP** modules.

```
#ifndef _DCE_CIOP_SECURITY_IDL
#define _DCE_CIOP_SECURITY_IDL

#include <IOP.idl>

#pragma prefix "omg.org"

module DCE_CIOPSecurity {

    const IOP::ComponentId   TAG_DCE_SEC_MECH = 103;

    typedef unsigned short     DCEAuthorization;

    const DCEAuthorization    DCEAuthorizationNone = 0;
    const DCEAuthorization    DCEAuthorizationName = 1;
    const DCEAuthorization    DCEAuthorizationDCE = 2;
```

```
    struct DCESecurityMechanismInfo {
        DCEAuthorization                    authorization_service;
        sequence<IOP::TaggedComponent>     components;
    };
};
#endif /* _DCE_CIOP_SECURITY_IDL */
```

A.19 Values for Standard Data Types

A number of data types in this specification allow an extensible set of values, so the user can add values as required to meet his own security policies. However, if all users defined their own values, portability and interoperability would be seriously restricted.

Therefore, some standard values for certain data types are defined. These include the values that identify:

- Security attributes (privilege and other attribute types)
- Rights families
- Audit event families and types
- Security mechanism types as used in the **IOR** (and **Vault**, etc.)

Rights families and audit event families are defined as an **ExtensibleFamily** type. This has a family definer value registered with OMG and a family id defined by the family definer. Security attribute types also have family definers. Family definers with values **0** - **7** are reserved for OMG. The family value **0** is used for defining standard types (e.g., of security attributes).

A.11.1 Attribute Types

"Application Developer's Interfaces" on page 15-88 defines an attribute structure for privilege and other attributes. This includes:

- A family, as previously described.
- An attribute type. Users may add new attribute types. Two standard OMG families are defined: the family of privilege attributes (**family = 1**), and the family of other attributes (**family = 0**). Types in these families are listed in the following table.
- An optional defining authority. This indicates the authority responsible for defining the value within the attribute type. Some policies demand that multiple sources of values for a given attribute type be supported (e.g., a policy accepting attribute values defined outside the security domain). These policies give rise to a risk of value clashes. The defining authority field is used to separate these values. When not present (i.e., **length = 0**), the value defaults to the name of the authority that issued the attribute.
- An attribute value. The attribute value is defined as a **sequence<octet>**, which someone who understands that attribute type can decipher.

*Table 15-25*Attribute Values

Attribute	**Type Value**	**Meaning**
Privilege Attributes (family 1)		**All privilege attributes are used for access control**
Public	1	The principal has no authenticated identity
AccessId	2	The identity of the principal used for access control
PrimaryGroupId	3	The primary group to which the principal belongs
GroupId	4	A group to which the principal belongs
Role	5	A role the principal takes
AttributeSet	6	An identifier for a set of related attributes, which a user or application can obtain
Clearance	7	The principal's security clearance
Capability	8	A capability
Other Attributes (family = 0)		
AuditId	1	The identity of the principal used for auditing
AccountingId	2	The id of the account to be charged for resource use
NonRepudiationId	3	The id of the principal used for non-repudiation

A.11.2 Rights Families and Values

Administration is simplified by defining rights that provide access to a set of operations, so the administrator only needs to know what rights are required, rather than the semantics of particular operations.

Rights are grouped into families. Only one rights family is defined in this specification. The family definer is OMG (value **0**) and the family id is CORBA (value **1**). Other families may be added by vendors or users.

The following values are specified for the standard CORBA rights family.

*Table 15-26*CORBA Rights Family Values

Right	**Meaning**
"**get**"	Used for any operation on the object that does not change its state
"**set**"	For operations on an object that changes its state
"**manage**"	For operations on the attributes of the object, not its state
"**use**"	For operations on an object that may change the overall state of the system, but not the state of the object itself

A.11.3 Audit Event Families and Types

Events, like rights, are grouped into families as defined in "Application Developer's Interfaces" on page 15-88.

Only one event family is defined in this specification. This has a family definer of OMG (value **0**) and family of SYSTEM (value **1**) and is used for auditing system events. All events of this type are audited by the object security services, or the underlying security services they use. Some of these events must be audited by secure object systems conforming to Security Functionality Level 1 (though in some cases, the event may be audited by underlying security services). Other event types are identified so that, if produced, a standard record is generated, so that audit trails from different systems can more easily be combined. System audit events are specified in Table 15-27.

*Table 15-27*System Audit Events

Event Name	**Value**	**Whether Mandatory**	**Meaning- and Event-Specific Data**
AuditPrincipalAuth	1	Yes	Authentication of principals, either via the principal authentication interface or underlying security services
AuditSessionAuth	2	Yes	Security association/peer authentication
AuditAuthorization	3	Yes	Authorization of an object invocation (normally using an Access Decision object)
AuditInvocation	4	No	Object invocation (i.e., the request/reply)
AuditSecEnvChange	5	No	Change to the security environment for this client or object (e.g., set_security_features, override_default_credentials)
AuditPolicyChange	6	Yes	Change to a security policy (using the administrative interfaces in Section 15.6, Administrator's Interfaces)
AuditObectjCreation	7	No	Creation of an object
AuditObject Destruction	8	No	Destruction of an object
AuditNonRepudiation	9	No	Generation or verification of evidence

Application audit policies are expected to use application audit families.

A.11.4 Security Mechanisms

The security specification allows use of different mechanisms for security associations. These are used in the Interoperable Object Reference and also on the interface to the Vault.

Mechanism ids are formed by stringifying the integer value of the corresponding mechanism tag value. For example, the mechanism id of mechanism type SPKM_1 is the string “15”, which is the string representation of the mechanism tag value defined in the SECIOP module above as TAG_SPKM_1_SEC_MECH.

Following this rule, the currently defined mechanism ids are:

*Table 15-28*Mechanism Ids

Mechanism Name	**Mechanism Tag**	**Mech Id**	**Base Mech**
SPKM_1	TAG_SPKM_1_SEC_MECH	“15”	SPKM
SPKM_2	TAG_SPKM_2_SEC_MECH	“16”	SPKM
KerberosV5	TAG_KerberosV5_SEC_MECH	“17”	KerberosV5
CSI_ECMA_Secret	TAG_CSI_ECMA_Secret_SEC_MECH	“18”	CSI_ECMA
CSI_ECMA_Hybrid	TAG_CSI_ECMA_Hybrid_SEC_MECH	“19”	CSI_ECMA
CSI_ECMA_Public	TAG_CSI_ECMA_Public_SEC_MECH	“21”	CSI_ECMA

Cryptographic profile ids are the stringified form of the value of the cryptographic profile constant. For example, the id of the cryptographic profile **MD5_RSA** is the string “**20**”. The cryptographic profile ids currently defined are:

*Table 15-29*Cryptographic Profile Ids

Profile Name	**Profile Id**	**Base Mech**
MD5_RSA	“20”	SPKM
MD5_DES_CBC	“21”	SPKM
DES_CBC	“22”	SPKM
MD5_DES_CBC_SOURCE	“23”	SPKM
DES_CBC_SOURCE	“24”	SPKM
DES_CBC_DES_MAC	“10”	KerberosV5
DES_CBC_MD5	“11”	KerberosV5
DES_MAC	“12”	KerberosV5
MD5	“13”	KerberosV5
FullSecurity	“1”	CSI_ECMAS
NoDataConfidentiality	“2”	CSI_ECMA
LowGradeConfidentaility	“3”	CSI_ECMA
AgreedDefault	“5”	CSI_ECMA

A complete mechanism type (used for **MechanismType** parameters) consists of a mechanism id with zero, one or more comma-separated cryptographic profiles appended to it. For example, the mechanism type "**15,20**" represents **SPKM_1** mechanism with **MD5_RSA** cryptographic profile.

Appendix B Relationship to Other Services

B.1 Introduction

This appendix describes the relationship between Object Services and Common Facilities and the security architecture components, if they are to participate in a consistent, secure object system.

B.2 General Relationship to Object Services and Common Facilities

In general, Object Services and Common Facilities, like any application objects, may be unaware of security, and rely on the security enforced automatically on object invocations. As for application objects, access to their operations can be controlled by access policies as described in "Security Reference Model" on page 15-18, "Application Developer's Interfaces" on page 15-88, and elsewhere.

An Object Service or Common Facility needs to be aware of security if it needs to enforce security itself. For example, it may need to control access to functions and data at a finer granularity than at object invocation, or need to audit such activities. The way it can do this is described in "Security Reference Model" on page 15-18. Existing Object Services should be reviewed to see if such access control and auditing is required.

If an Object Service or Common Facility is required to be part of a more secure system, some assurance of its correct functioning, if security-relevant, is needed, even if it is not responsible for enforcing security itself. See Appendix D, "Guidelines for a Trustworthy System" on page 15-329, for guidelines on this matter.

Where an Object Service is called by an ORB service as part of object invocation in a secure system, there is a need to ensure security of all the information involved in the invocation. This requires ORB Services to be called in the order required to provide the specified quality of protection. For example, the Transaction Service must be invoked first to obtain the transaction context information before the whole message is protected for integrity and/or confidentiality.

In the following sections, we provide an initial estimation of the relationship between Security Service and other existing services and facilities.

B.3 Relationship with Specific Object Services

B.3.1 Naming Service

For security, the object must be correctly identified wherever it is within the distributed object system. The Naming Service must do this successfully in an environment where an object name is unique within a naming context, and name spaces are federated. (However, to provide the required proof of identity, objects, and/or the gatekeepers which give access to them will be authenticated using a separate Authentication Service.) See Appendix Section D.6.3, “Basis of Trust,” on page 15-347, for additional information about the relationship between security and names.

B.3.2 Event Service

The implementation of a Security Audit Service may involve the use of Event Service objects for the routing of both audits and alarms.

However, this is only possible if the Event Service itself is secure in that it protects the audit trail from modification and deletion. It must also be able to guard against recursion if it audits its own activities.

B.3.3 Persistent Object Service

No explicit use is made of this service. Audit trails may be saved using this service, in which case the implementation of the Persistent Object Service must ensure that data stored and retrieved through it is not tampered with by unauthorized entities. If it is used in the implementation of Security Service or by a secure application, it must follow the guidelines in Appendix D, “Guidelines for a Trustworthy System” on page 15-329.

B.3.4 Time Service

The Security Service uses the data types for time, timestamps, and time intervals as defined by the Time Service, so that applications can readily use the Time Service-defined interfaces to manipulate the time data that the Security Service uses. The interfaces of Security Service do not explicitly pass any interfaces defined in the Time Service.

B.3.5 Other Services

The other services are not used explicitly. If any of them are used in the implementation of Security Service or by a secure application, it must be verified that the service used follows the guidelines in Appendix D, “Guidelines for a Trustworthy System” on page 15-329.

B.4 Relationship with Common Facilities

Because Management Services have been identified as Common Facilities in the Object Management Architecture, only minimal, security-specific administration interfaces are specified here. When Common Facilities Management services are specified, they will need to take into account the need for security management and administration identified in this specification. Also, such management services will themselves need to be secure.

This specification adds certain basic interfaces to CORBA, which form the basis for the minimal policy administration-related interfaces and functionality that has been provided. Future management facilities are expected to build upon this foundation.

Appendix C Conformance Details

C.1 CORBA Security Feature Packages

C.1.1 Main Security Functionality

There are two possible levels.

- ***Level 1***: This provides a first level of security for applications unaware of security, and for those that have limited requirements to enforce their own security in terms of access controls and auditing.
- ***Level 2***: This provides more security facilities, and allows applications to control the security provided at object invocation. It also includes administration of security policy, allowing applications administering policy to be portable.

C.1.2 Security Functionality Options

These are functions expected to be required in several ORBs, so they are worth including in this specification, but are not generally required enough to form part of one of the main security functionality levels previously specified. There is only one such option in this specification.

- ***Non-Repudiation***: This provides generation and checking of evidence so that actions cannot be repudiated.

C.1.3 Security Replaceability

This specification is designed to allow security policies to be replaced. The additional policies must also conform to this specification. This includes, for example, new Access Polices. Security Replaceability specifies if and how the ORB fits with different security services. There are two possibilities.

- ***ORB Services replaceability***: The ORB uses interceptor interfaces to call on object services, including security ones. It must use the specified interceptor interfaces and call the interceptors in the specified order. An ORB conforming to this does not include any significant security-specific code, as that is in the interceptors.
- ***Security Service replaceability***: The ORB may or may not use interceptors, but all calls on security services are made via the replaceability interfaces specified in Section 15.7, "Implementor's Security Interfaces," on page 15-156. These interfaces are positioned so that the security services do not need to understand how the ORB works, so they can be replaced independently of that knowledge.

An ORB that supports one or both of these replaceability options is said to be Security-Ready (i.e., supports no security functionality itself, but is ready to have security added).

Note – Some replaceability of the security mechanism used for secure associations may still be provided if the implementation uses some standard generic interface for security services such as GSS-API.

C.1.4 Secure Interoperability using SECIOP

An ORB supporting this can generate/use security information in the IOR and can send/receive secure requests to/from other ORBs using the GIOP/IIOP protocol with the security (SECIOP) enhancements defined in Section 15.9, "Secure Inter-ORB Protocol (SECIOP)," on page 15-204, providing they can both use the same underlying security mechanism and algorithms for security associations.

Common Secure Interoperability (CSI) Feature packages: These feature packages each provide different levels of secure interoperability. There are three functionality levels for Common Secure Interoperability (CSI).

All levels can be used in distributed secure CORBA-compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages - for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

An ORB conforming to CSI level 2 can support all the security functionality described in the CORBA Security specification. Facilities are more restricted at levels 0 and 1. The three levels are:

1. ***Identity-based policies without delegation (CSI level 0)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

2. ***Identity-based policies with unrestricted delegation (CSI level 1)***: At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in "Overview of Delegation Schemes" on page 15-31.)

3. ***Identity- & privilege-based policies with controlled delegation (CSI level 2)***: At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

An ORB that interoperates securely must provide at least one of the CSI packages. For the definitive statement on conformance requirements see Appendix C, "Conformance Details" on page 15-315.

C.1.5 Common Security Protocol packages

The choice of protocol to use depends on the mechanism type required and the facilities required by the range of applications expected to use it. Common Security Protocols define the details of the tokens in the IIOP and SECIOP messages as applicable. Four protocols are defined:

1. ***SPKM Protocol***: This protocol supports identity-based policies without delegation (CSI level 0) using public key technology for keys assigned to both principals and trusted authorities. The SPKM protocol is based on the definition in [20]. The use of SPKM in CORBA interoperability is based on the SECIOP extensions to IIOP.

2. ***GSS Kerberos Protoco****l*: This protocol supports identity-based policies with unrestricted delegation (CSI level 1) using secret key technology for keys assigned to both principals and trusted authorities. It is possible to use it without delegation (providing CSI level 0). The GSS Kerberos protocol is based on the [12] which itself is a profile of [13]. The use of Kerberos in CORBA interoperability is based on the SECIOP extensions to IIOP.

3. ***CSI-ECMA protocol***: This protocol supports identity- and privilege-based policies with controlled delegation (CSI level 2). It can be used with identity, but no other privileges and without delegation restrictions if the administrator permits this (CSI level 1) and can be used without delegation (CSI level 0). For keys assigned to principals, it has the following options:
   - It can use either secret or public key technology.
   - It uses public key technology for keys assigned to trusted authorities.

   The CSI-ECMA protocol is based on the ECMA GSS-API Mechanism as defined in ECMA 235, but is a significant subset of this - the SESAME profile as defined in [16]. It is designed to allow the addition of new mechanism options in the future; some of these are already defined in ECMA 235. The use of CSI-ECMA in CORBA interoperability use the SECIOP extensions to IIOP.

   DCE-CIOP*:* An ORB supporting this option provides secure interoperability using DCE Security together with the Security extensions to DCE-CIOP.

4. ***SSL protocol***: This protocol supports identity-based policies without delegation (CSI level 0). The SSL protocol is based on the definition in [21]. The use of SSL in CORBA interoperability does not depend on the SECIOP extensions to IIOP.

An ORB that interoperates securely must do so using one of these protocol packages. For the definitive statement on conformance requirements see Appendix E, "Conformance Statement" on page 15-352.

C.2 Conformance Requirements

An ORB must meet the following requirements to claim conformance to the CORBA Security specification:

- To claim conformance to the **CORBA Security** interfaces it must support the following feature packages:
  - **Security Functionality Level 1**.
- To claim conformance to **CORBA Secure Interoperability** it must support the following feature packages:
  - **Secure Interoperability using SECIOP**.
  - **CSI Level 1**.
  - **GSS Kerberos Protocol using MD5 Cryptographic profile**.
- Conformance to any of the other feature packages may be claimed in addition to the base conformance specified in the previous bullet item, by providing the interfaces, facilities and support for protocols specified in that package, as described further in the following sections.

The conformance statement required for a CORBA Security-conformant implementation is defined in Section Appendix F, "Facilities Not in This Specification. Appendix F includes two checklists, one for functionality and the other for interoperability, which can be completed to show what the ORB conforms to; they are reproduced next. A main security functionality level must always be specified. Functional Options, Security Replaceability, and Secure Interoperability should be indicated by checking the boxes corresponding to the function supported by the ORB.

*Table 15-30*CORBA Security Functionality Checklist

Main Functionality		**Functionality Options**	**Security Replaceability**		
Level 1	**Level 2**	**Non-Repudiation**	**ORB Services**	**Security Services**	**Security-Ready**

*Table 15-31*CORBA Secure Interoperability Checklist

Interop	**IIOP**							**DCE-**
Level	**SECIOP**						**SSL**	**CIOP**
	SPKM		**Kerberos**	**CSI-ECMA**				
	SPKM 1	SPKM 2		Private	Public	Hybrid		
Level 0								
Level 1	**XXXX**	**XXXX**					**XXX**	
Level 2	**XXXX**	**XXXX**	**XXXXX**				**XXX**	

C.3 Security Functionality Level 1

Security Functionality Level 1 provides:

- A level of security functionality available to applications unaware of security. (It will, of course, also provide this functionality to applications aware of security.) This level includes security of the invocation between client and target object, simple delegation of client security attributes to targets, ORB-enforced access control checks, and auditing of security-relevant system events.
- An interface through which a security-aware application can retrieve security attributes, which it may use to enforce its own security policies (e.g., to control access to its own attributes and operations).

C.3.6 Security Functionality Required

An ORB supporting Level 1 security functionality must provide the following security features for all applications, whether they are security-aware or not.

- Allow users and other principals to be authenticated, though this may be done outside the object system.
- Provide security of the invocation between client and target object including:
  - Establishment of trust between them, where needed. At Level 1, this may be supported by ORB-level security services or can be achieved in any other secure way. For example, it could use secure lower-layer communications. Mutual authentication need not be supported.
  - Integrity and/or confidentiality of requests and responses between them.
  - Control of whether this client can access this object. At this level, access controls can be based on “sets” of subjects and “sets” of objects. Details of the Access Policy and how this is administered are not specified.
- At an intermediate object in a chain of calls, the ability to be able to either delegate the incoming credentials or use those of the intermediate object itself.
- Auditing of the mandatory set of system’s security-relevant events specified in Appendix A, Consolidated OMG IDL. In some cases, the events to be audited may occur, and be audited, outside the object system (for example, in underlying security services). In this case, the conformance statement must identify the product responsible for generating the record of such an event (or choice of product, for example, when the ORB is portable to different authentication services). At this level, auditing of object invocations need not be selectable. However, it must be possible to ensure that certain events are audited (see Section A.11.3, “Audit Event Families and Types,” on page 15-309, for the list of mandatory events).

Note – For security-aware applications, it must also make the privileges of authenticated principals available to applications for use in application access control decisions.

These facilities require the ORB and security services to be initialized correctly. For example, the Current object at the client must be initialized with a reference to a credentials object for the appropriate principal.

C.3.7 Security Interfaces Supported

Security interfaces available to applications may be limited to:

- **get_service_information** providing security options and details (see Section 15.5.2, "Finding Security Features," on page 15-90).
- **get_attributes** on **Current** (see Interfaces under "Security Operations on Current" on page 15-105).

No administrative interfaces are mandatory at this level.

C.3.8 Other Security Conformance

An ORB providing Security Functionality Level 1 may also conform to other security options. For example, it may also:

- Support some of the Security Functionality Options specified in Section C.5, "Security Functionality Optional Packages," on page 15-322.
- Provide security replaceability using either of the replaceability options.
- Provide secure interoperability, though in this case, will need to provide security associations at the ORB level (not lower-layer communications) as the protocol assumes security tokens are at this level.

C.4 Security Functionality Level 2

This is the functionality level that supports most of the application interfaces defined in "Application Developer's Interfaces" on page 15-88, and the administrative interfaces defined in "Administrator's Interfaces" on page 15-130. It provides a competitive level of security functionality for most situations.

C.4.1 Security Functionality Required

An ORB that supports Security Functionality Level 2 supports the functionality in Security Level 1 previously defined, and also:

- Principals can be authenticated outside or inside the object system.
- Security of the invocation between client and target objects is enhanced.
    - Establishment of trust and message protection can be done at the ORB level, so security below this (for example, in the lower layer communications) is not required (though may be used for some functions).
    - Further integrity options can be requested (e.g., replay protection and detection of messages out of sequence) but need not be supported.

- The standard **DomainAccessPolicy** is supported for control of access to operations on objects.
- Selective auditing of methods on objects is supported.

- Applications can control the options used on secure invocations. It can:
  - Choose the quality of protection of messages required (subject to policy controls).
  - Change the privileges in credentials.
  - Choose which credentials are to be used for object invocation.
  - Specify whether these can just be used at the target (e.g., for access control) or whether they can also be delegated to further objects.
- No further delegation facilities are mandatory, but the application can request "composite" delegation, and the target can obtain all credentials passed, in systems that support this. Note that "composite" here just specifies that both received credentials and the intermediate's own credentials should be used. It does not specify whether this is done by combining the credentials or linking them.
- Administrators can specify security policies using domain managers and policy objects as specified in "Administrator's Interfaces" on page 15-130. The security policy types supported at Level 2 are all those defined in Section 15.6, "Administrator's Interfaces," on page 15-130 except non-repudiation. The standard policy management interfaces for each of the Level 2 policies is supported.
- Applications can find out what security policies apply to them. This includes policies they enforce themselves (e.g., which event types to audit) and some policies the ORB enforces for them (e.g., default qop, delegation mode).
- ORBs (and ORB Services, if supported) can find out what security policies apply to them. They can then use these policy objects to make decisions about what security is needed (check if access is permitted, check if auditing is required) or get the information needed to enforce policy (get QOP, delegation mode, etc.) depending on policy type.

As at Level 1, these facilities require the ORB and security services to be initialized correctly.

C.4.2 Security Interfaces Supported

Interfaces supported at this level are:

- All application interfaces defined in Section 15.5, "Application Developer's Interfaces," on page 15-88, except those in Section 15.5.11, "Non-repudiation," on page 15-121.
- All security policy administration interfaces defined in Section 15.6, Administrator's Interfaces (except those for the non-repudiation policy).

Note that some of these interfaces may raise a CORBA::NO-IMPLEMENT exception, as not ORBs conforming to Level 2 Security need implement all possible values of all parameters. This will happen when:

- A privilege attribute is requested of a type that is not supported (attribute types supported are defined in Appendix Section A.10, “General Security Data Module,” on page 15-283).
- A delegation mode is requested, which is not supported.
- A communication direction for association options is requested, which is not supported.

C.4.3 Other Security Conformance

An ORB providing Security Functionality Level 2 may also conform to other security options. For example, it may also:

- Support some of the Security Functionality Options specified in Appendix Section C.6, “Security Replaceability,” on page 15-323.
- Provide security replaceability, using either of the replaceability options.
- Provide secure interoperability.

C.5 Security Functionality Optional Packages

An ORB may also conform to optional security functionality defined in this specification. Only one optional facility is specified: non-repudiation.

Also, some requirements on conformance of additional facilities are specified.

C.5.1 Non-repudiation

Security Functionality

An ORB conforming to this must support the non-repudiation facilities for generating and verifying evidence described in “The Model as Seen by the Objects Implementing Security” on page 15-79. Note that these use **NRCredentials**, the attributes in which may be the same as in the credentials used for other security facilities. Where non-repudiation is supported, the credentials acquired from the environment or generated by the authenticate operation must be able to support non-repudiation.

Security Operations Supported

The following operations must be supported. All are available to applications. They are:

- **set_/get_NR_features** as defined in Section 15.5.11, “Non-repudiation,” on page 15-121.
- **generate_token, verify_evidence, form_complete_evidence** and **get_token_details of NRCredentials** object as defined in “Non-repudiation” on page 15-121.

- Use of **set/get_credentials on Current** specifying the type of credentials to be used is **NRCredentials**.
- **NRPolicy** object with associated interfaces as in Section 15.6.7, “Non-repudiation Policy Management,” on page 15-153.

Fit with Other Security Conformance

Non-repudiation requires use of credentials; thus it can only be used with ORBs, which support some of the interfaces defined in Security Functionality level 2. However, conformance to all of Security Functionality Level 2 is not a prerequisite for conformance to the non-repudiation security functionality option.

Secure interoperability as defined in Appendix Section C.7, “Secure Interoperability,” on page 15-325, is not affected by non-repudiation. The evidence may be passed on an invocation as a parameter to a request, but the ORB need not be aware of this.

The current specification does not specify interoperability of evidence (i.e., one non-repudiation service handling evidence generated by another).

C.5.2 Conformance of Additional Policies

This specification is designed to allow security policies to be replaced. The additional policies must also conform to some of the interfaces in this specification if they are used to replace the standard policies automatically enforced on object invocation.

The case described next is for the addition of a new Access Policy which can be used for controlling access to objects automatically, replacing the standard **DomainAccessPolicy**.

Clearly, other policies can be replaced. For example, the audit policy could be replaced by one that used different selectors, or the delegation policy could be replaced by one that supported more advanced features.

C.6 Security Replaceability

This specifies how an ORB can fit with security services, which may not come from the same vendor as the ORB. As explained above, there are two levels where this can be done (apart from any underlying APIs used by an implementation).

C.6.1 Security Features Replaceability

Conformance to this allows security features to be replaced.

If it is provided without conformance to the ORB Service replaceability option (see Appendix Section C.6.2, “ORB Services Replaceability,” on page 15-324), it requires the ORB to have a reasonable understanding of security, handling credentials, etc., and knowing when and how to call on the right security services.

Support for this replaceability option requires an ORB (or the ORB Services it uses) to use the implementation-level security interfaces as defined in Section 15.7, “Implementor’s Security Interfaces,” on page 15-156. This includes:

- The **Vault**, **Security Context**, **Access Decision**, **Audit**, and **Principal Authentication** objects defined in Section 15.7.2, “Implementation-Level Security Object Interfaces,” on page 15-161.
- Certain features of the CORBA Core needed for ORB Service Replaceability can be found in the *Common Object Request Broker: Architecture and Specification*.

C.6.2 ORB Services Replaceability

Conformance to this allows an ORB to know little about security except which interceptors to call in what order. This is intended for ORBs, which may use different ORB services from different vendors, and require these to fit together. It therefore provides a generic way of calling a variety of ORB Services, not just security ones. It also assumes that any of these services may have associated policies, which control some of their actions.

Support for this replaceability option requires an ORB to:

- Use the Interceptor interfaces defined in the Interceptor chapter of the *Common Object Request Broker: Architecture and Specification* to call security interceptors defined in Section 15.7.1, “Security Interceptors,” on page 15-156, in the order defined there.
- Use the **get_policy** operation (and the associated security policy operations such as **access_allowed, audit_needed** defined in “Access Control” on page 15-117 and Section 15.5.7, “Security Audit,” on page 15-113 respectively, for access control and audit and also **get_association_options** and **get_delegation_mode** defined in Section 15.6.6, “Secure Invocation and Delegation Policies,” on page 15-147, for association options, quality of protection of messages, and delegation).

C.6.3 Security-Ready for Replaceability

An ORB is Security-Ready for Replaceability if it does not provide any security functionality itself, but does support one of the security replaceability options.

Security Functionality Required

An ORB that is Security-Ready does not have to provide any security functionality, though must correctly respond to a request for the security features supported.

Security Interfaces Supported

- **get_service_information** operation providing security options and details (see Section 15.5.2, “Finding Security Features,” on page 15-90).

- **get_current** operation to obtain the Current object for the execution context (see the ORB Interface chapter of the *Common Object Request Broker: Architecture and Specification*).

Other Security Conformance

An ORB that is Security-Ready for replaceability supports one of the replaceability options. This should be done in such a way that the ORB can work without security, but can take advantage of security services when they become available. So it calls on the replaceability interfaces correctly (using dummy routines to replace security services when these are needed, but not available).

C.7 Secure Interoperability

The definition of secure interoperability in this specification specifies that a conformant ORB can:

- Generate, and take appropriate action on, Interoperable Object References (IORs), which include security tags as specified in Section 15.8.4, “CORBA Interoperable Object Reference with Security,” on page 15-178.
- Transmit and receive the security tokens needed to establish security associations, and also the protected messages used for protected requests and responses once the association has been established according to the protocol defined in Section 15.9, “Secure Inter-ORB Protocol (SECIOP),” on page 15-204

Note – A Security-Ready ORB (i.e., with no built-in security functionality) may, by additions of appropriate security services, conform to secure interoperability.

For ORBs to interoperate securely, they must choose to use the same mechanism, algorithms, etc., (or use a bridge between them, if available). A set of standard security mechanisms and algorithms are described in subsections.

C.7.1 Standard Secure Interoperability

An ORB that conforms to this must support the security-enhanced IOR defined in Section 15.8.4, “CORBA Interoperable Object Reference with Security,” on page 15-178, and also GIOP/IIOP protocol with the SECIOP enhancements as defined in Section 15.9, “Secure Inter-ORB Protocol (SECIOP),” on page 15-204.

As for CORBA 2, this may be done by immediate bridges or half bridges. (However, use of half bridges implies more complex trust relationships, which some systems may not be able to support.) This allows a large range of security mechanisms to be used.

C.7.2 Common Secure Interoperability Levels

There are three functionality levels for Common Secure Interoperability (CSI). An example of the difference in use of the three levels is explained in Appendix Section C.7.2, “Common Secure Interoperability Levels,” on page 15-326.

All levels can be used in distributed secure CORBA-compliant object systems where clients and objects may run on different ORBs and different operating systems. At all levels, security functionality supported during an object request includes (mutual) authentication between client and target and protection of messages - for integrity, and when using an appropriate cryptographic profile, also for confidentiality.

An ORB conforming to CSI level 2 can support all the security functionality described in this specification. Facilities that are supportable at levels 0 and 1 are more restricted. The three levels are:

1. Identity-based policies without delegation (CSI level 0)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target, and this cannot be delegated to further objects. If further objects are called, the identity will be that of the intermediate object, not the initiator of the chain of object calls.

Access and audit policies at this level are based on the identity of the immediate invoker. So access and audit policies in encapsulated objects which depend on the initiator of the chain, can only be used at the point of entry to the object system, not in further objects encapsulated by it.

As the attributes of principals are not delegated, environments should not be trusted to pass on principal information which should be controlled.

Examples of applications which can use level 0 facilities are wrapped legacy applications and telephone switches. If a CSI level 0 ORB also supports non-repudiation, it can also be used for other types of applications such as electronic funds transfer.

2. Identity-based policies with unrestricted delegation (CSI level 1)

At this level, only the identity (no other attributes) of the initiating principal is transmitted from the client to the target. The identity can be delegated to other objects on further object invocations, and there are no restrictions on its delegation, so intermediate objects can impersonate the user. (This is the impersonation form of simple delegation defined in Section 15.3.6, “Delegation,” on page 15-30.)

Access and audit policies at this level can be based on the identity of the initiating principal or immediate invoker, depending on the delegation policy.

As delegation is not restricted, once an initiator has delegated his identity, it must trust the objects it calls not to abuse its delegated rights to act as the initiator. In practice, this will limit the type of environment in which level 1 should be used to relatively closed environments.

An example of an application environment which can use level 1 facilities is a back office system protected by firewalls where identity-based policies are acceptable.

3. Identity- & privilege-based policies with controlled delegation (CSI level 2)

At this level, attributes of initiating principals passed from client to target can include separate access and audit identities and a range of privileges such as roles and groups. Delegation of these attributes to other objects is possible, but is subject to restrictions, so the initiating principal can control their use. Optionally, composite delegation is supported, so the attributes of more than one principal can be transmitted. Therefore, it provides interoperability for ORBs conforming to all CORBA Security functionality.

Access and audit policies are based on the attributes of initiating principals. At this level, a wider range of policies can be supported (e.g., role-based access controls and mandatory access controls using the initiating principal's security clearance).

At this level, an initiator needs to trust those targets which it has allowed to use its attributes not to abuse these. It does not have to trust these targets not to delegate the attributes outside the trusted set of targets, as the delegation controls can be used to prevent this.

This level can be used for a wide range of applications in large enterprise and inter-enterprise networks.

C.7.3 SECIOP-Hosted Interoperability Mechanisms

The following conformance can be claimed:

- SPKM at level 0 by providing the specified CSI level using the SPKM protocol (mechanism **SPKM_1** and optionally also **SPKM_2**).
- **KerberosV5** at level 0 or 1 by providing the specified CSI level using the Kerberos protocol.
- CSI-ECMA Public Key at level 0, 1, or 2 by providing the specified level of CSI functionality using the CSI-ECMA protocol with the public key option (mechanism **CSI_ECMA_Public**).
- CSI-ECMA Secret Key at level 0, 1, or 2 by providing the specified CSI level using the CSI-ECMA protocol with the secret key option (mechanism **CSI_ECMA_Secret**).
- CSI-ECMA Hybrid at level 0, 1, or 2 by providing the specified CSI level using the CSI-ECMA protocol with the hybrid key option (mechanism **CSI_ECMA_Hybrid**).

In addition, a conformant ORB must specify all the cryptographic profiles it supports.

C.7.4 Secure Interoperability with SSL

Conformance can be claimed for CORBA Security based on SSL by providing CSI level 0 functionality using SSL on IIOP using any of the cryptographic profiles defined in [21]. A conformant ORB must specify which of the cryptographic profiles are supported by it.

C.7.5 Secure Interoperability with DCE-CIOP

An ORB that conforms to this must conform to Standard Secure Interoperability using GIOP/IIOP as described in Appendix Section C.7.1, "Standard Secure Interoperability," on page 15-325, and also support secure interoperability using DCE-CIOP as defined in Section 15.15, "DCE-CIOP with Security," on page 15-273.

Both the Kerberos V5-based SECIOP Security and DCE Security must be supported for this option. Any version of DCE up to and including DCE 1.1 is supported; the DCE interfaces and protocols are specified in [5].

Appendix D Guidelines for a Trustworthy System

D.1 Introduction

This appendix provides some general guidelines for helping ORB implementors produce a trustworthy system. The intention is to have all information related to trustworthiness and assurance in this appendix, to explain how the specification has taken into account the requirements for assurance, and also to show how conformant implementations can have different levels of assurance.

The remainder of the introduction first provides the rationale for including these guidelines in the specification, and then gives some background on trustworthiness and assurance. Appendix Section D.2, "Protecting Against Threats," on page 15-331, describes the threats and countermeasures relevant to a CORBA security implementation. Appendix Section D.3 through D.6 provide the architecture and implementation guidelines for each security object model described in Section 15.4, "Security Architecture," on page 15-45.

D.1.1 Purpose of Guidelines

The security standards proposed in this specification have been deliberately chosen to allow flexibility in the security features, which can be provided. This specification can support significantly different security policies and mechanisms for security functions such as access control, audit, and authentication. However, there is an overall security model which applies, whatever the security policy. This is described in the earlier sections of this specification.

There is also flexibility in the level of security assurance, which can be provided, conforming to this model and these standards. This appendix describes the trustworthiness issues underlying the security model and interfaces described earlier in this specification, and provides implementation guidance on what components of the architecture need to be trusted and why. Note that trust requirements assume conformance to all of the security models, including the implementor's view, as the implementation affects trustworthiness. If a CORBA security implementation conforms to the security features replaceability level, but not the ORB services one, any requirements on ORB services will apply to the ORB. Trustworthiness will also depend on several other implementation choices, such as the particular security technology used.

D.1.2 Trustworthiness

Before an enterprise places valuable business assets within an IT system, enterprise management must decide whether the assets will be adequately protected by the system. Management must be convinced that the particular system configuration is sufficiently *trustworthy* to meet the security needs of the enterprise environment. Security trustworthiness is thus the ability of a system to protect resources from exposure to misuse through malicious or inadvertent means.

The basis for trust in distributed systems differs from host-centric stand-alone systems largely for two reasons. First, the assignment of trust in a distributed system is not isolated to a single global system mechanism. Second, the degree of trust in elements of distributed systems (particularly distributed *object* systems) may change dynamically over time, whereas in host-centric systems trustworthiness is typically static. In many cases, trust in distributed systems must be seen in the context of mutual suspicion.

D.1.3 Assurance

Assurance is a qualitative measure of trustworthiness; assurance is the confidence that a system meets enterprise security needs. The qualitative nature of assurance means that enterprises may have different assurance guidelines for an equivalent level of confidence in security. Some organizations may need extensive evaluation criteria, while other organizations need very little evidence of trustworthiness.

It is necessary to set a context by which CORBA developers and end-users of the CORBA Security specification may evaluate the level of security to meet their needs. A single overall trust model that underlies the security reference model and architecture (as described elsewhere in this specification) can set this context for closed systems, but it is unlikely that a single trust model exists for the diversity of open distributed systems likely to populate the distributed object technology world.

To support a balanced approach, assurance arguments should be assembled from a set of system building blocks. Concepts of system composition and integration should allow the assurance analysis to be tailored to specific user requirements. Assurance evidence should be carefully packaged to best support enterprise decision-makers during the security trade-off process.

The security object models defined by the CORBA Security specification are the basis for the necessary building blocks. The trust guidelines described in "Guidelines for Structural Model" on page 15-335, provide constraints on how these components may relate.

The relationship between assurance and security provides the foundation for the overall security model. The key characteristic is balance. Balanced assurance promotes the use of assurance arguments and evidence appropriate to the level of risk in the system components.

Basic system building blocks, such as those in the CORBA Security specification previously noted, are critical to developing balanced assurance. For example, confidentiality is of most importance to a classified intelligence or military system, whereas data integrity may be of more importance in a computer patient record system. The former relies on assurance in the underlying operating system, where the latter focuses security in application software.

D.2 Protecting Against Threats

An enterprise needs to protect its assets against perceived threats using appropriate security measures. This specification addresses security in distributed object systems, so it focuses on the threats to assets, software, and data in such systems.

An enterprise may want to assess the risk of a security breach occurring against the damage which will be done if it does occur. The enterprise can then decide the best trade-off between the cost of providing protection from such threats and any performance degradation this causes, against the probability of loss of assets. This specification allows options in how security is provided to counter the threats. However, it is expected that many enterprises will not undertake a formal risk assessment, but rely on a standard level of protection for most of their assets, as identified by industry or government criteria. This section describes CORBA-specific security goals, the main distributed system threats, and protection against them. The discussion does not emphasize generic issues of threats and countermeasures, but instead concentrates on issues that are unique to the CORBA security architecture.

D.2.1 Goals of CORBA Security

The overall goals of the CORBA security architecture were described in “Introduction to Security” on page 15-2. CORBA security is based on the four fundamental objectives of any secure system:

- Maintain confidentiality of data and/or system resources.
- Preserve data and/or system integrity.
- Maintain accountability.
- Assure data/system availability.

Many of the goals described in Section 15.1, “Introduction to Security,” on page 15-2 are relevant to any IT system that is targeted at large-scale applications. However, some security goals described are specific to the CORBA security architecture. These goals deserve special attention because they bring attention to potential threats that may not be encountered in typical architectures. CORBA-specific security goals include:

- Providing security across a heterogeneous system where different vendors may supply different ORBs.
- Providing purely object-oriented security interfaces.
- Using encapsulation to promote system integrity and to hide the complexity of security mechanisms under simple interfaces.
- Allowing polymorphic implementations of objects based on different underlying mechanisms.
- Ensuring object invocations are protected as required by the security policy.
- Ensuring that the required access control and auditing is performed on object invocation.

The discussion of the architecture and implementation guidelines in Appendix Section D.3, "Guidelines for Structural Model," on page 15-335, addresses the mechanisms used to ensure these CORBA-specific security goals, as well as many other generic security issues.

D.2.2 Threats

The CORBA security model needs to take into account all potential threats to a distributed object system. It must be possible to set a security policy and choose security services and mechanisms that can protect against the threats to the level required by a particular enterprise.

A security *threat* is a potential system misuse that could lead to a failure in achieving the system security goals previously described. Section 15.1, "Introduction to Security," on page 15-2, provided an overview of security threats in a distributed object system. These threats and related attacks include:

- **Information compromise** - the deliberate or accidental disclosure of confidential data (e.g., masquerading, spoofing, eavesdropping).
- **Integrity violations** - the malicious or inadvertent modification or destruction of data or system resources (e.g., trapdoor, virus).
- **Denial of service** - the curtailment or removal of system resources from authorized users (e.g., network flooding).
- **Repudiation of some action** - failure to verify the actual identity of an authorized user and to provide a method for recording the fact (e.g., audit modification).
- **Malicious or inadvertent misuse** - active or passive bypassing of controls by either authorized or unauthorized users (e.g., browsing, inference, harassment).

The threats described above give rise to a wide variety of attacks. Most if not all the threats that pertain to host-centric systems are pertinent to distributed systems. Furthermore, it appears likely that the wide distribution of resources and mediation in truly distributed systems will not only exacerbate the strain on host-centric security services and mechanisms in use today on client/server systems, but also engender new forms of threat.

Threats may be of different strengths. For example, accidental misuse of a system is easier to protect against than malicious attacks by a skilled hacker. This specification does not attempt to counter all threats to a distributed system. Those that should be countered by measures outside the scope of this specification include:

- Denial of service, which may be caused by flooding the communications with traffic. It is assumed that the underlying communications software deals with this threat.
- Traffic analysis.
- Inclusion of rogue code in the system, which gives access to sensitive information. (This affects the build and change control process.)

D.2.3 Vulnerabilities of Distributed Object-Oriented Systems

Vulnerabilities are system weaknesses that leave the system open to one or more of the threats described above. Information systems are subject to a wide range of vulnerabilities, a number of which are compounded in distributed systems. These vulnerabilities often result from deliberate or unintentional trade-offs made in system design and implementation, usually to achieve other more desirable goals such as increased performance or additional functionality.

Classes of vulnerabilities include:

- An authorized user of the system gaining access to some information which should be hidden from that user, but has not been properly protected (e.g., access controls have not been properly set up or the store occupied by one object has not been cleared out when another reuses the space).
- A user masquerading as someone else, and so obtaining access to whatever that user is authorized to do, resulting in actions being attributed to the wrong person. In a distributed system, a user may delegate his rights to other objects, so they can act on his behalf. This adds the threat of rights being delegated too widely, again, causing a threat of unauthorized access.
- Controls that enforce security being bypassed.
- Eavesdropping on a communication line giving access to confidential data.
- Tampering with communication between objects: modifying, inserting, and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

System data as well as business data must be protected. For example:

- If a principal’s credentials are successfully obtained by an unauthorized user, they could be used to masquerade as that principal.
- If the security-sensitive information in the security context between client and target object is available to an unauthorized user, confidential messages can be read, and it may be possible to modify and resend integrity-protected messages or send false messages without this being detected.

As described earlier, system threats and vulnerabilities are compounded by the complexities of distributed object-based systems. Some of the inherent characteristics of distributed object systems that make them particularly vulnerable include:

- **Dynamic Systems** -- Distributed object systems are always changing. New components are constantly being added, deleted, and modified. Security policies also may be dynamically modified as enterprises change. Dynamic systems are inherently complex, and thus security may be difficult to ensure. For example, in a large distributed object system it will be difficult to update a security policy atomically. While an administrator installs a new policy on some parts of the system, other parts of the system still may be using the old version of the policy. These potential inconsistencies in policy enforcement could lead to a security failure.

- **Mutual Suspicion** -- In a large distributed system, some system components will not trust others. Mistrust could occur at many layers within the architecture: principals, objects, administrators, ORBs, and operating systems may all have varying degrees of trustworthiness. In this environment, there is always the potential to inadvertently place unjustified trust in some system component, thus exposing a vulnerability. Although there are many mechanisms (e.g., cryptographic authentication) to ensure the identity of a remote component, the system security architecture must be carefully structured to ensure that these checks are always performed.
- **Multiple Policy Domains** -- Distributed object systems that interconnect many enterprises are likely to require many different security policy domains, each one enforcing the security requirements of its organization. There is no single security policy and enforcement mechanism that is appropriate for all businesses. As a result, security policies must be able to address interactions across policy domain boundaries. Defining the appropriate policies to enforce across domains may be a difficult job. Mismatched policies could lead to vulnerabilities.
- **Layering of Security Mechanisms** -- Distributed object systems are highly layered, and the security mechanisms for those systems will be layered as well. Complex, potentially nondeterministic interactions at the boundary of the layers is another area for vulnerabilities to occur. A hardware error, for example, could cause security checking code in the ORB to be bypassed, thus violating the policy. The complexity of the layering is further compounded in systems where security enforcement is widely distributed; that is, there is no clear security perimeter containing only a small amount of simple functionality.
- **Complex Administration** -- Finally, large geographically distributed object systems may be difficult to administer. Security administration requires the cooperation of all the administrators, who even may be mutually suspicious. All of the issues listed above lead to complex, error-prone administration. An innocent change to a principal's access rights, for example, could expose a serious vulnerability.

D.2.4 Countermeasures

Some threats are common across most distributed secure systems, so they should be countered by standard security features of any OMA-compliant secure systems. However, the level of protection against these threats may vary. Complete protection is almost impossible to achieve. Most enterprises will want a balance between a level of protection against threats which are important to them, and the cost in performance and use of other resources of providing that level of protection.

A number of measures exist for countering or mitigating the effects of the above threats/attacks. Countering these threats requires the use of the security object models described in this specification. Relevant features of the object models include the following:

- Authentication of principals proves who they are, so it is possible to check what they should be able to do. This check can be performed at both client and target object, as the client principal's credentials can be passed to the server.

- Authentication between clients and target objects allows them to check that they are communicating with the right entities.
- Security associations can protect the integrity of the security information in transit between client and target object (e.g., credentials, keys) to prevent theft and replay, and keep the keys used for protecting business data confidential.
- Business data can be integrity-protected in transit so any tampering is detected using the message protection ORB services. (This includes detecting extra or missing messages, and messages out of sequence.)
- Unauthorized access to objects is protected using access controls.
- Misuse of the system can be detected using auditing.
- Segregating (groups of) applications from each other and security services from applications can prevent unauthorized access between them.
- Bypassing of security controls is deterred by use of a Trusted Computing Base (TCB), where security is automatically enforced during object invocation.

Assurance arguments and evidence are frequently founded on the concept of a TCB, which mediates security by segregating the security-relevant functions into a security kernel or reference monitor.

A traditional monolithic TCB approach is not suitable for the open, multiuser, multiple environment situations in which most CORBA users reside. In many cases, for example, secure interoperability of CORBA applications and ORBs may be based on mutual suspicion. TCB scalability issues also argue against typical TCB approaches. Given the complexity of distributed systems, it is not clear whether centralized access mediation is possible in the presence of distributed data and program logic.

Traditional TCB approaches also do not adequately address application security requirements, particularly for many commercial applications. Applications common to the CORBA world such as general purpose DBMSs, financial accounting, electronic commerce, or horizontal common facilities will have many security requirements in addition to those that can be enforced by a central underlying TCB.

Despite the limitations of the traditional TCB, we use the concept of a *distributed TCB* in the assurance discussions of the next section. The concept of a distributed TCB is the collection of objects and mechanisms that must be trusted so that end-to-end security between client and target object is maintained. However, note that depending on the assurance requirements of a particular CORBA security architecture, sensitive data may still be handled by "entrusted" ORB code. Thus, our informal use of the distributed TCB concept may not correspond to other existing models for network TCBs, particularly for minimal assurance commercial CORBA security applications.

D.3 Guidelines for Structural Model

This section provides architecture and implementation guidelines for the structural model of the CORBA security architecture described in Section 15.4, "Security Architecture," on page 15-45. The security functions provided in the model and the basis for trust are described in this section.

D.3.1 Security Functions

Figure 15-63 outlines interactions during a normal use of the system. It gives a simple case, where the application is unaware of security except for calling a security service such as audit. The security interactions include those seen by application objects and secure object system implementors.

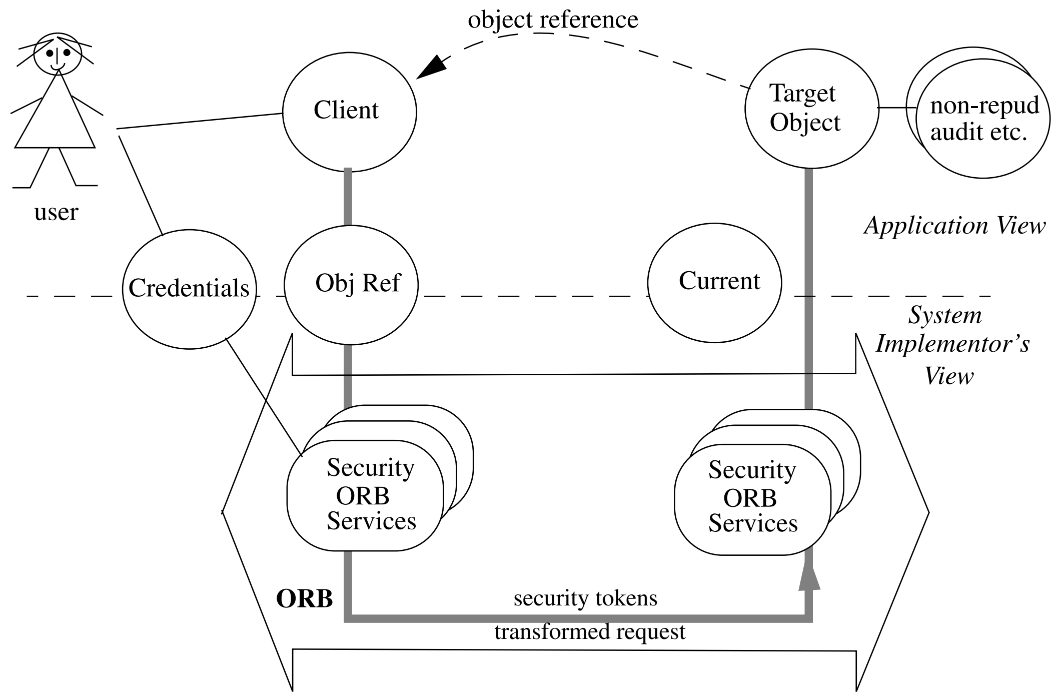


Figure 15-63 Normal System Interactions

This diagram is the basis for the discussions of security functions in each of the security object models described next.

D.3.2 Basis of Trust

Enterprise management is responsible for setting the overall security policies and ensuring system enforcement of the policies.

The system developer and systems integrators must provide a system that supports the required level of assurance in the core security functionality. Generally application developers cannot be expected to be aware of all potential threats to the system, and to put the right countermeasures in place.

Higher levels of security may require the code enforcing it to be formally evaluated according to security criteria such as those of the US TCSEC or European ITSEC.

Distributed Trusted Computing Base

The key security functionality in the system is enforced transparently to the application objects so that it can be provided for application objects which are security-unaware. This key functionality is contained in the distributed TCB of the system. It is therefore responsible for ensuring that:

- Users cannot invoke objects unless they have been authenticated (unless the security policy supports unauthenticated, guest access for some services).
- Security policies on access control, audit, and security association are enforced on object invocation. This includes policies for message protection, both confidentiality (ensuring confidential data cannot be read) and integrity (ensuring any corruption of data in transit is detected).
- A principal's credentials are automatically transferred on object invocation if required, so the access control and other security policies can be enforced at the server object.
- Application objects which do not trust each other cannot interfere with each other.
- The security policy between different security policy domains is suitably mediated.
- The security mechanisms themselves cannot be tampered with.
- The security policy data cannot be changed except by authorized administrators.
- The system cannot be put into an undefined or insecure state as a result of the operation of nonprivileged code.

The distributed TCB also needs to provide the required information so that applications can enforce their own security policies in a way that is consistent with the domain security policy.

.

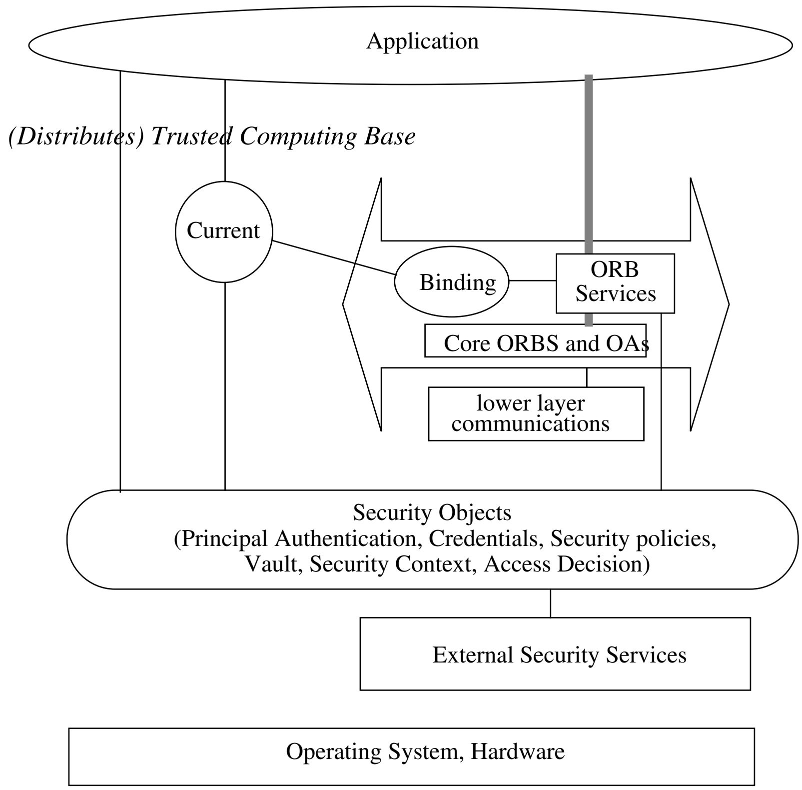


Figure 15-64 Distributed TCB

The TCB in an OMA-compliant secure system is normally distributed and includes components as follows.

- The distributed core ORBs and associated Object Adapters - Core ORBs are trusted to function correctly and call the ORB Security Services correctly in the right order, but do not need to understand what these do. Object Adapters are trusted to utilize the operating system facilities to provide the required protection boundaries between components in line with the security policy.
- The associated ORB Services - ORB Services other than security are trusted similarly to the ORB. ORB Security Services are used to provide the required security on object invocation.
- Related objects - ORB Services use objects such as the binding and Current to find which security is required.
- Security objects - Security objects include those available to applications such as Principal Authentication and Credentials and those called by security interceptors (Vault, Security Context, Access Decision, and Security Audit). These are trusted to function correctly to enforce security in line with the security policy and other requirements.

- Any external security services used by the security services, as part of enforcing the security policy.
- The supporting operating systems. These are trusted to ensure that objects (in different trust domains) cannot interfere with each other (using protection domains). The security services should also ensure that the security information driving the security policy (such as the credentials and security contexts) is adequately protected from the application objects using such features.
- Optionally, lower-layer communications software. However, this does not generally need to be particularly secure (at least for normal commercial security) as protection of data in transit is done by the security association and message protection interceptors, which are independent of the underlying communication software.

A distributed system may be split into domains, which have different security policies. These domains may include ORBs and ORB Services with different levels of trust. Trust between domains needs to be established, and an interdomain policy between them enforced. The ORB security services (and external security services that these call) to provide this interdomain working are part of the distributed TCB. Note, therefore, that the parts of this TCB in different domains may have different levels of trust.

Note – Note that application objects may enforce their own security polices, if these are consistent with the policy of the security domain. However, failure to enforce these securely will affect only the applications concerned and any other application objects that trusted them to perform this function.

Protection Boundaries

The general approach is to establish **protection boundaries** around groups of one or more components, which are said to belong to a corresponding **protection domain**. Components belonging to a protection domain are assumed to trust each other, and interactions between them need not be protected from each other, whereas interactions across boundaries may be subject to controls. Protection Boundaries and Domains are a lower-level concept than Environment Domains; they are the fundamental protection mechanism on which higher levels are built.

At a minimum, it must be possible to create protection boundaries between:

- Application components that do not trust each other.
- Components that support security services and other components.
- Components that support security services and each other.

Controlled Communications

As well as providing protection boundaries, it is necessary to provide a controlled means of allowing particular components to interact across protection boundaries (for example, an application invoking a Security Object (explicitly), or an interceptor (implicitly).

It must not be possible for applications to bypass security services which enforce security policies. It is therefore necessary to ensure that the components supporting those services are always invoked when required. This is achieved by using both protection boundaries and controlled communications to ensure that client requests (and server responses) are routed via the components (interceptors and Security Objects), which implement the security services.

Figure 15-65 illustrates the segregation of components implementing security services into separate protection domains from application components; the only means of communication between components is via controlled communication paths.

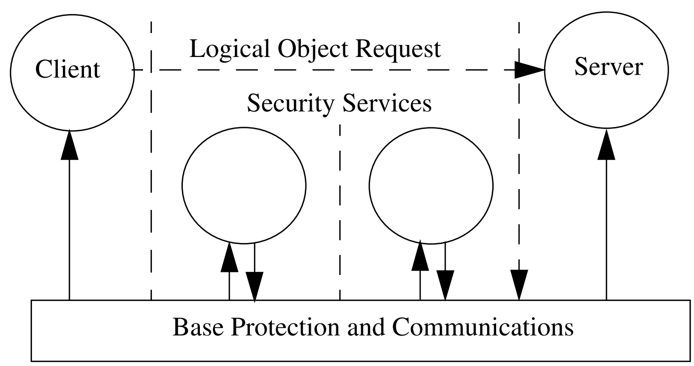


Figure 15-65 Base Protection and Communications

In implementation terms, components could, for example, be executed in separate processes, with process boundaries acting as protection boundaries. Alternatively, security services could be executed in-process with (i.e., in the same address space as) corresponding client and server application components, provided that they are adequately protected from each other -- for example, by hardware-supported multilevel access control mechanisms).

Figure 15-66 shows two examples of protection boundaries. In the first example, the boundaries between components might be process boundaries. In the second example, ORB and security components might be protected from applications by memory protection mechanisms (e.g., kernel and user spaces) and client and server components might be protected from each other by physical separation.

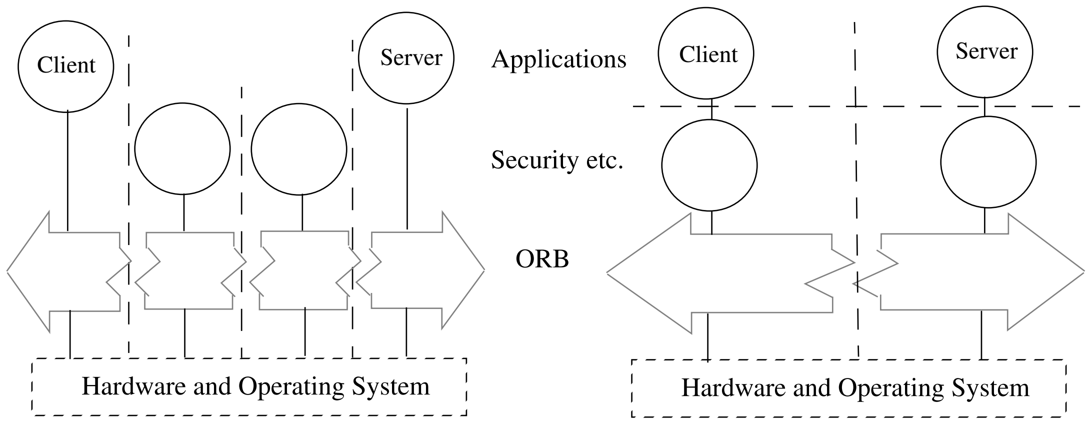


Figure 15-66 Protection Boundaries

D.3.3 Construction Options

For some systems, the TCB in domains of the distributed system may need to meet security evaluation criteria for both functionality and assurance (in the correctness and effectiveness of the security functionality) as defined in TCSEC, ITSEC, or other security evaluation criteria.

The split into components previously described allows a choice over the way the system is constructed to meet different requirements for assurance and performance.

This section describes three options for how the system may be constructed, as follows:

- A commercial system where all applications are generated using trusted tools.
- A commercial system with limited security requirements.
- A higher security system.

Note – These are just examples to show the type of flexibility provided by the security model. It is not expected that any implementation will provide all the options implied by these.

Example Using Trusted Generation Tools and ORBs

If all applications are generated using trusted tools, applications can be trusted not to interfere with other components in the same environment. Therefore there is no need to provide protection boundaries between different application objects or between application objects and the underlying ORB.

If the ORB and ORB Services are also trusted, there may be no need to provide a protection boundary between the ORB and the underlying security services and objects. It may well be acceptable to run them all in the same process, relying on the trust between the components, rather than more rigidly enforced boundaries.

However, if the application generation tools and the ORB are less trusted than the security services, then there may need to be a protection boundary to prevent access to security-sensitive information in the Credentials, Security Context, and Vault objects.

Commercial System with Limited Security Requirements

Some systems may not contain very sensitive business information, so enterprises may not be prepared to pay for a high-level of security. They may also know that the probability of serious malicious attempts to break the system is low, and decide that protecting against such attempts is not worth the cost. They may also choose not to sacrifice performance for better levels of security.

In many systems, applications are generated using tools that are not particularly trusted. For example, using a C compiler, it would be possible to write an application that can read, or even alter, any information within the same protection domain. Theoretically, providing good security implies putting protection boundaries between application objects, and between applications and the ORB and Security Services.

The security model allows environment domains to be defined, where enforcement of policy can be achieved by means local to the environment. For example, objects in the same identity domain can share a security identity. Applications belonging to environment domains may trust each other not to interfere with each other, so they can be put in the same protection domain.

It may also be acceptable to run (part of) the ORB in the same protection domain as the application objects. This assumes that an interface boundary between applications and the ORB is sufficient protection from accidental damage (the probability of an application corrupting an ORB being low in a commercial system). Even if the application does corrupt the ORB, damage is limited, as the ORB does not handle security-sensitive data.

In some commercial systems, it may also be acceptable to run some of the security services in the same protection domain as the application and ORB. The chance of these being accidentally (or maliciously) corrupted may be low, so it may be acceptable to risk a failure to enforce the access control policy because the Access Decision object is corrupt.

However, it will often be desirable to protect the state information of security objects, which contain very sensitive security information from the applications.

Higher Security System

In a security system requiring high assurance, different security policies may be used. For example, label-based access controls may be used and these may be mandatory (set under administrator's controls) and not changeable by application objects.

Stronger protection boundaries are also likely to be needed, allowing:

- Individual applications to be protected from each other. Even if environment domains are used, the size of the domain is likely to be smaller.
- The ORB and ORB Services to be protected from the application.
- The core security objects, which contain security-sensitive information such as keys to be protected from applications and ORBs, etc.
- Particular secure objects (e.g., the Access Decision objects) to be separate from others, as they may have been written by someone less trusted than those who wrote, for example, the Security Context objects.

D.3.4 Integrity of Identities (Trojan Horse Protection)

In traditional procedural systems, protecting the integrity of an identity is straightforward; programs are stored in files, which are protected against modification by operating system access control mechanisms. When invoked, programs run inside a process whose address space is protected by operating system memory protection mechanisms. Programs load code in fairly predictable ways.

Since this specification does not mandate which entities have identities, implementors have a wide variety of choices; identities may be associated, for example, with the following:

- Object instances
- Servers
- Object adaptors
- Address spaces

If identities are associated with object instances, precautions are necessary to prevent object instance code from being modified by other code (which may have no identity, or a different identity) in the instance's address space.

Servers may permit dynamic instantiations of previously unknown classes into their address spaces. This makes it difficult to determine what code is running under an identity if identities are associated with servers; this in turn makes it difficult to determine whether a server identity can be "trusted." Identified servers must therefore be provided with some way of controlling what code can run under their identities.

Observing the following guidelines will help to ensure integrity of identities.

- Code running under one identity must not be permitted to modify code running under another identity without passing an authorization check.
- It must be possible for an identified "entity" to control which code runs within the scope of its identity.

D.4 Guidelines for Application Interface Model

This section provides architecture and implementation guidelines for the application interface model of the CORBA security architecture described in Section 15.4, “Security Architecture,” on page 15-45. The security functions provided in the model and the basis for trust are described in this section.

D.4.1 Security Functions

Logging onto the System

When a user or other principal wants to use a secure object system, it authenticates itself and obtains credentials. These contain its certified identity and (optionally) privilege attributes, and also controls where and when they can be used. This principal information is integrity-protected and it should be possible to ascertain what security service certified them.

Walkthrough of Secure Object Invocation

The following is a walkthrough of what happens when a client invokes a target object.

- The client invokes the object using its object reference. The ORB Security Services are transparent to the client and application object and use the security information with the object reference and the security policy to decide on the security facilities required. There are separate ORB Services for security associations, message protection, and access control on object invocation, but the audit service can be called by any or none of these according to security policy.

  The client and target object establish the required level of trust in each other, transmitting security tokens to each other to provide the required degree of proof. For example, they may or may not require mutual authentication. It is expected that most security mechanisms will provide options here, though the details of how they do this, and the form of tokens used, is mechanism-dependent.

  The principal’s credentials are normally passed transparently from client to target object. These should be protected in transit from theft and replay as well as for integrity of the information itself (though some security mechanisms may not support this). The **Vault** object will validate these, checking that it trusts who certified them, as well as whether they are still intact.

  Different ORB services may be called at the target end. For example, access control is normally called at the server, rather than the client.

- Once the security association has been established between client and target object, the request can be passed using the message protection interceptor to protect it. This should be able to provide integrity and/or confidentiality protection. It should also be able to provide continuous authentication, as the messages will be protected using keys only known to this client and server (or the trust group for the target object).

- The application object may also call security services for access control and audit. These will use the security information available from the environment to identify the initiating principal and its privileges.
- This application object may now act as a client, and call further objects. It may delegate the client's credentials or use its own (or use both). However, there may be constraints on whether the client's credentials can be delegated. For example, a particular principal's credentials may be constrained to particular groups of objects.

D.4.2 Basis of Trust

Users have some trust in application objects, and application objects have some trust in other objects. Both may:

- Trust application objects to perform the business functions.
- Have limited trust in some applications, or domains of the distributed system, so they may restrict which of their privilege attributes are available to these objects.
- Want to restrict the extent to which their credentials can be propagated if at all.
- Have to prove their identity to the system so it can enforce access on their behalf, unless they are only going to access publicly available services.

Both users and applications trust the underlying system to enforce the system security policy, and therefore protect their information from unauthorized access and corruption.

D.5 Guidelines for Administration Model

This section provides architecture and implementation guidelines for the administration model of the CORBA security architecture described in Section 15.4, "Security Architecture," on page 15-45. The security functions provided in the model and the basis for trust are described.

D.5.1 Security Functions

Object and Object Reference Creation

When an object is created in a secure object system, the security attributes associated with it depend on the security policies associated with its domain and object type, though the object may be permitted to change some of these. These attributes control what security is enforced on object invocation (for example, whether access control is needed and, if so, the Access Decision object to be used; the minimum quality of protection required).

The object reference for a such an object is extended to include some security information. For example, it may contain:

- An extended identity. This includes the object identity as normal in an object reference. However, it will also contain the identity of the trust domain, if the object belongs to one. Small objects, which are dynamically created and do not need to be protected from each other, will normally share a trust domain. There could also be a node identity.
- Security policy attributes required by the object when invoked by a client such as the minimum quality of protection of data in transit.
- The security technology it supports. It may also contain some mechanism-specific information such as its public key, if public key technology is being used, and particular algorithms used.

Much of the information is just “hints” about which security is required, and will be verified by the ORB services supporting the target object, so it does not need protecting.

D.5.2 Basis of Trust

Authorization Policy Information

Domain objects may store policy information inside their own encapsulation boundaries, or they may store it elsewhere (for example, authorization policy information could be encapsulated in the state data of the protected objects themselves, or it could be stored in a procedural Access Control Manager whose interfaces are accessible to Domain objects). Wherever authorization policy information is stored, it must be protected against modification by unauthorized users.

Authorization policy information must be modifiable only by authorized administrators.

Audit Policy Information and Audit Logs

Audit policy information is security-sensitive and must be protected against unauthorized modification. Audit logs are security-sensitive and may contain private information; they should be viewed and changed only by authorized auditors.

- Audit policy information must be modifiable only by authorized audit administrators.
- Audit logs must be protected against unauthorized examination and modification.

D.6 Security Object Implementation Model

D.6.1 Guidelines

This section provides architecture and implementation guidelines for the security object implementation model of the CORBA security architecture described in Section 15.4, "Security Architecture," on page 15-45. The security functions provided in the model and the basis for trust are described.

D.6.2 Security Functions

The distributed core ORBs, object adapters, ORB security services, and security objects provide the underlying implementation to support the application and administration interfaces.

D.6.3 Basis of Trust

Target Object Identities

CORBA objects do not have unique identities; for this reason, when objects that are not associated with a human user authenticate themselves in a secure CORBA system, they use "security names." Successful authentication to a target object indicates that it possesses the authentication data (perhaps a cryptographic key), which is presumed to be known only to the legitimate owner of the security name. An object's security name may be included in references to that object as a "hint." The question "how do applications know that the security-name hint is reliable?" naturally arises.

The answer is as follows:

- If the **EstablishTrustinTarget** security feature is specified, then the security services defined in this specification will authenticate the target security name found in the target object reference. The semantics of this authentication operation include an assumption that the security name in the reference corresponds to an identity that the user is willing to trust to provide the target object's implementation. There is no way for the security services to test this assumption.
- If your implementation provides a trusted source of object references, then everything will work properly. If you do not have a source of trusted object references, the specification provides a **get_security_names** operation on the object reference through which applications can retrieve the target's security name and perform any tests, which may help satisfy them of its validity.

CORBA object references can circulate very widely; for example, they can be "stringified" and then (potentially) copied onto a piece of paper. Implementations with very high integrity requirements could ensure that references are trustworthy by providing a trustworthy service that generates references and cryptographically signs the contents, including the target security name.

Assumptions about Security Association Mechanisms

Implementation of a secure CORBA system requires use of security mechanisms to enforce the security with the required degree of protection against the threats. For example, cryptographic keys are normally used in implementing security, for functions such as authenticating users and protecting data in transit between objects. However, different security mechanisms may use different types of cryptographic technology (e.g., secret or public key) and may use it in different ways when, for example, protecting data in transit. These cryptographic keys have to be managed, and again, the way this is done is mechanism-specific.

A full analysis of how well an implementation counters the threats requires knowledge of the security mechanisms used. However, this specification does not dictate that a particular mechanism is used.

It does assume that the security mechanisms used for authentication and security associations can provide the relevant security countermeasures listed in Appendix Section D.2.4, "Countermeasures," on page 15-334. These are expected to be provided by a number of security mechanisms, which will be available for protecting secure object systems. Therefore, the analysis of threats and the trust model assume this facility level.

It would be possible to use a security mechanism that does not provide some of these facilities (for example, mutual authentication, or even to switch this off to improve performance in systems that can provide it). However, if such a system is used, it will be vulnerable to more threats.

Invoking Special Objects

Some of the objects described in this specification are **locality-constrained** objects, which bypass the normal invocation process and therefore are not subject to the security enforced by the ORB services. The **Current** object (used, for example, by the target object to obtain security information about the client) is of this type. Protection of these objects is provided by other means, for example, using protection boundaries previously described.

D.6.4 Basis For ORB Assurance

The ORB must function correctly (e.g., when enforcing security policy on object invocation and object creation as defined in this specification). Likewise the underlying host platforms must function correctly in their provision of the security mechanisms employed, and relied upon, by the ORB. Both must do this to the level of assurance specified in its Conformance Statement (which is described in Appendix Section Appendix E, "Conformance Statement," on page 15-352). This section identifies many of the most critical design considerations related to providing these assurances in a DOC system.

Isolating Security Mechanisms

Figure 15-67 depicts how security functionality and trust is distributed throughout the architecture.

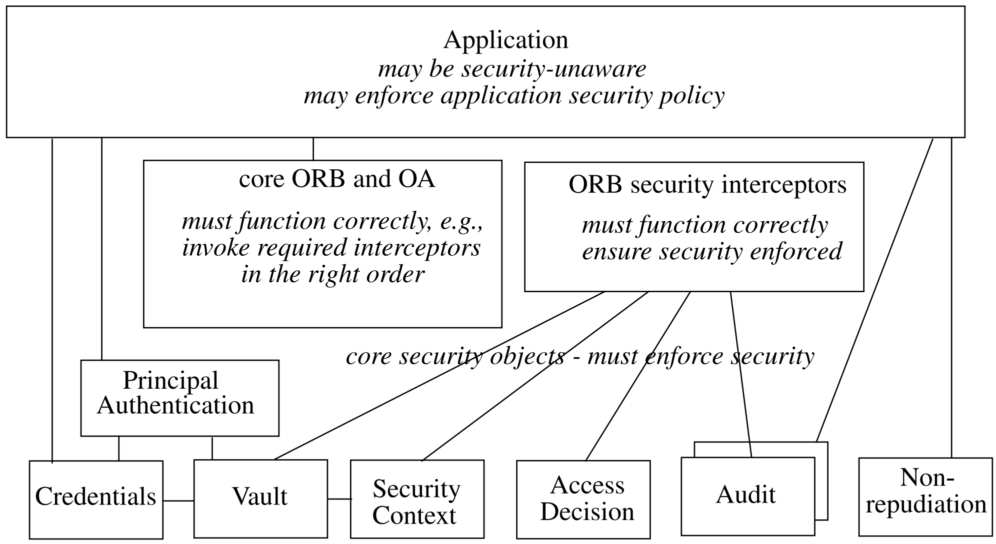


Figure 15-67 Distribution of Security Functionality and Trust

The split of security objects is designed to reduce (as much as possible) the amount of security-sensitive information, which must be visible to applications and ORBs.

- Only log-in applications (where provided) need to handle secrets such as passwords, and then only briefly during authentication.
- Cryptographic keys and other security-sensitive information about principals are held with Credentials objects. References to Credentials objects are visible to applications so they can invoke operations on them to, for example, reduce privileges in the credentials before calling an object. However, no operations on the Credentials provide visibility of security information such as keys.
- Security information used to protect application data in transit between objects is held in Security Context objects, which are not visible to applications at all. (Target applications can ask for attributes associated with an incoming invocation using the Current object.)

Security objects such as Credentials, Security Context, and Access Decision objects are also not used directly by the core ORB, only by the security interceptors. Therefore the core ORB needs to be trusted to call the interceptors correctly in the right order, but does not need to understand security or have access to the security-sensitive information in them.

The split also is intended to isolate components which may be replaced to change security policy or security mechanisms. For example, to replace the access control policy, the Access Decision objects need to be changed. However, the access control interceptor will remain responsible for finding and invoking the right Access Decision object. To replace the security mechanisms for security association, only the **Vault** and associated Security Context objects need to be replaced.

Integrity of the ORB and Security Service Objects

Security in a CORBA environment depends on the correct operation of the ORB and Security Services. In order for these mechanisms to operate correctly, the following rules must be followed:

- The ORB and **Vault** code must not be modifiable by unauthorized users or processes.
- The ORB must protect all messages, according to policy, using the message protection interfaces.
- The ORB must always check the client's authorization before dispatching a client's message to a protected object.

Safeguarding the Object Environment

To guard against unauthorized modification of the ORB and security services, implementors should use Operating System protection mechanisms to isolate the ORB and Security Service objects from untrusted applications and user code.

Note that some modifications of ORB or **Vault** code may not compromise system integrity. For example, in a CORBA implementation, which relies on third-party authentication and does not share **Vault** or ORB objects between processes, corruption of the client-side **Vault** (or ORB) by user-written code may not compromise system security. (This is because the client-side ORB and **Vault** in a third-party-based system may, depending upon the implementation, contain only information that the user is entitled to know and change anyway. In this case, nothing the user can do to information on his machine will enable him to deceive the third-party authentication server about his identity and credentials.)

Safeguarding the Dispatching Mechanism

To ensure that the ORB always checks the client's authorization before dispatching a client's message to a protected object, ORB implementors should follow one of the following rules:

- Eliminate "direct dispatching" mechanisms (which permit clients to dispatch messages directly to target objects without going through the ORB).
- Permit "direct dispatching" only after checking authorization and issuing "restricted object references" to client objects. A "restricted object reference" is one that grants access only to those methods of the target object, which the client is authorized to invoke.

Safeguarding Information in Shared Vault Objects

Vault objects encapsulate identity-specific, security-sensitive information (for example, cryptographic keys associated with Security Context objects). If code owned by one principal can penetrate a **Vault** object and examine or modify another principal's information, security can be compromised.

In an implementation that does not permit sharing of **Vault** objects by multiple identities, this problem does not arise. However, if **Vault** objects are accessible to and encapsulate information about multiple identities, the following guidelines should be observed:

- Do not permit a **Vault** object, which encapsulates one principal's Security Contexts, to exist in the same address space as code running under a different principal's identity.
- If a **Vault** object contains Security Contexts for two different principals, ensure that no principal is able to obtain or use another principal's Security Contexts.

Appendix E Conformance Statement

E.1 Introduction

A secure object system, like any secure system, should not only provide security functionality, but should also provide some assurance of the correctness and effectiveness of that functionality.

Each OMG-compliant security-ready implementation must therefore include in its documentation a conformance statement describing:

- The product's supported security functionality levels and options, security replaceability, and security interoperability, as described in Appendix C, "Conformance Details" on page 15-315.
- The vendor's assurance argument that demonstrates how effectively the product provides its specified security functionality and security policies.
- Constraints on the use of the product to ensure security conformance.

The vendor provides the conformance statement so that a potential product user can make an informed decision on whether a product is appropriate for a particular application. Ordinary descriptive documentation is not required as part of an OMG-compliant product. However, because the CORBA security specification provides a general security framework rather than a single model, there are many different kinds of secure ORB implementations that conform to the framework. For example, some systems may have greater flexibility and support customized security policies, while other systems may come with a single built-in policy. Some systems may strive for a high level of security assurance, while others provide minimal assurance. The conformance statement will help the user understand the security features provided by the product.

Some products will undergo an independent formal security evaluation (such as ones meeting the ITSEC or TCSEC). The OMG security conformance statement does not take the place of a formal evaluation, but may refer to formal assurance documentation, if it exists. When formal evaluations are not required (often the case in commercial systems), it is expected that the product's security conformance statement along with supporting product documentation will provide an adequate description of security functionality and assurance.

E.2 Conformance Template Overview

The following template specifies the contents for CORBA security conformance statements. Guidelines for using this template are provided in Section, Conformance Guidelines.

CORBA Security Conformance Statement

<date>

<product identification>

<vendor identification>

1. Introduction

1.1 Summary of Security Conformance

1.2 Scope of Product

1.3 Security Overview

2. Security Conformance

2.1 Main Security Functionality Level

2.2 Security Functionality Options

2.3 Security Replaceability

2.4 Secure Interoperability

3. Assurance

3.1 Philosophy of Protection

3.2 Threats

3.3 Security Policies

3.4 Security Protection Mechanisms

3.5 Environmental Support

3.6 Configuration Constraints

3.7 Security Policy Extensions

4. Supplemental Product Information

E.3 Conformance Guidelines

The guidelines in this section are intended to help the ORB implementor determine which information belongs in each section of the conformance statement. The statement will often be accompanied by product documentation to provide some of the information needed.

1. Introduction

1.1 Summary of Security Conformance

This section should give a summary of the security conformance provided by the product. The summary is in the form of a table with boxes that are ticked to show the relevant conformance.

*Table 15-32*CORBA Security Functionality Checklist

Main Functionality		**Functionality Options**	**Security Replaceability**		
Level 1	**Level 2**	**Non-Repudiation**	**ORB Services**	**Security Services**	**Security-Ready**

*Table 15-33*CORBA Secure Interoperability Checklist

Interop	**IIOP**							**DCE-**
Level	**SECIOP**						**SSL**	**CIOP**
	SPKM		**Kerberos**	**CSI-ECMA**				
	SPKM 1	SPKM 2		Private	Public	Hybrid		
Level 0								
Level 1	**XXXX**	**XXXX**					**XXX**	
Level 2	**XXXX**	**XXXX**	**XXXXX**				**XXX**	

For the main security functionality level, one of the boxes must be selected (either Level 1 or Level 2), though note that an ORB can be just Security-Ready, so does not support either of the main security functionality levels. For security functionality options, security replaceability, and secure interoperability, the appropriate boxes should be selected.

1.2 Scope of Product

This section should define what security components this product offers. Examples are:

- ORB plus all security services needed to support it plus other object services fitting with it and meeting the assurance criteria.
- Security-ready ORB.

- Security Services, which can be used with a security-ready ORB.

1.3 Security Overview

This section should give an overview of the product's security features.

2. Security Conformance

2.1 Main Security Functionality Level

This section should define which main security functionality level this product supports, Level 1 or Level 2.

This should also include any qualifications on that support. For example, any interpretation of the CORBA security specification and how it is supported, any bells and whistles around the published interfaces, and any limitations on support for this level.

As in the conformance-level descriptions, the description should be divided into:

- The security functionality provided by the product
- The application developer's interfaces
- The administrative interfaces

2.2 Security Functionality Options

This section should define which functionality options are provided, in particular the support for non-repudiation.

For non-repudiation, as this is a published interface in this specification, it should be accompanied by a qualification statement if needed, as for the main security functionality level.

2.3 Security Replaceability

This section should define whether the product supports replaceability of security services, ORB services, or neither.

This should also include any qualifications on that support. For example, any interpretation of the CORBA security specification and how it is supported, any bells and whistles around the published interfaces, and any limitations on support for this conformance option.

2.4 Secure Interoperability

This section should define whether the product supports SECIOP-based secure interoperability, DCE-CIOP-based interoperability, SSL-based interoperability, or none. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

2.5 Level of Interoperability

This section should specify what level of interoperability is supported by the ORB. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

2.6 Mechanism Profiles

This section should specify what mechanism and cryptographic profiles for interoperability are supported by the ORB. As with the previous sections, qualifications of the support, interpretations of the CORBA specification, and limitations should be included as needed.

3. Assurance

If the product already has supporting assurance documentation (for example, because it is being formally evaluated), much of this section may be satisfied by references to such documentation. Appendix E, Guidelines for a Trustworthy System, provides general discussions of many of the topics described here, particularly the basis of trust needed for each of the architecture object models.

3.1 Philosophy of Protection

Overview of supported security policies, security mechanisms and supporting mechanisms.

3.2 Threats

Description of specific threats intended to be addressed by the system security policy, as well as those not addressed.

3.3 Security Policies

Description of any predefined policies, including:

- Classes of entities (such as clients, objects) controlled by security policy
- Modes of access (conditions that allow active entities to access objects)
- Use of domains (policy, trust, technology)
- Requirements for authentication of principal, client, and target objects
- Requirements for trusted path between principals, clients, ORBs, and target objects
- Delegation model
- Security of communications
- Accountability requirements (audit, non-repudiation)
- Environmental assumptions of the policy (e.g., classes of users, LAN/WAN, physical protection)

3.4 Security Protection Mechanisms

- Rationale for approach
- Identification of components, which must function properly for security policies to be enforced
- Description of mechanisms used to enforce security policy
- How protection mechanisms are distributed in the architecture
- Why security mechanisms (such as access control) are always invoked and tamper-proof

3.5 Environmental Support

- How the underlying environment (such as operating systems, generation tools, hardware, network services, time services, security technology) are used in providing assurance
- How installation tools ensure secure configuration
- How security management and administration maintain secure configuration

3.6 Configuration Constraints

Constraints to ensure that system security assurance is preserved, for example:

- Requirements on use and development of: clients, target objects, legacy software
- Limitations on interoperability
- Required software and hardware configuration

3.7 Security Policy Extensions

- Supported security policy extensions, if applicable
- Limitations of extensions
- Requirements imposed on developers to ensure trustworthiness of policy extensions
- Supported interactions and compositions of security policies

4. Supplemental Product Information

Supplemental product information is included at the vendor's discretion. It can be used to describe, for example:

- Additional security features not covered by the CORBA Security specification
- The impact of security mechanisms on existing applications

Appendix F Facilities Not in This Specification

F.1 Introduction

Security in CORBA systems is a big subject which affects many parts of the Object Management Architecture. It was therefore decided to phase the specification in line with the priorities agreed to as part of the security evaluation criteria by the Security Working Group prior to the production of this specification.

This specification therefore includes the core security facilities and the security architecture to allow further facilities to be added. Priority has been given to those requirements most needed by commercial systems. Even with these limitations, the size of the specification is larger than desirable for OMG members to review easily or for vendors to implement.

Some of the facilities omitted from this specification are agreed to be required in some secure CORBA systems, and so they are expected to be added later, using the usual OMG process of RFPs to request their specification.

This appendix lists those security facilities which are not included in the specification, but left to later specifications, which may be in response to further RFPs for Object Services or Common Facilities.

F.2 Interoperability Limitations between Unlike Domains

Secure interoperability is included in this specification. This allows applications running under different ORBs in different domains to interoperate providing that:

- Both support and can use the same security mechanisms (and algorithms, etc.,) for authentication and secure associations (an ORB may support a choice of security mechanisms).
- Use of these between the domains will not contravene any government regulations on the use of cryptography.
- The security policies they support are consistent -- for example, use the same types for privileges which can be understood in both places.

Limitations in the specification which affect this type of interoperability are:

- The standard policies defined do not include specifying different policies when a client communicates with different domains (though it is possible to define specific policies to do this).
- There is no specification of the mapping policies required to translate attributes when crossing a domain boundary where these policies are inconsistent, and how these must be positioned, for example, to allow delegation of the mapped attributes. Again, such mapping policies are not prevented.
- In general, there is no specification of how federated policies are implemented.

- There is no specification of gateways to handle interoperability between security mechanisms. It is expected that only limited interoperability between particular security mechanisms will ever be provided, so this is not expected to be the subject of an RFP in the foreseeable future.

F.3 Non-Session-Oriented SECIOP Protocol

The SECIOP protocol defined in Section 15.9, "Secure Inter-ORB Protocol (SECIOP)," on page 15-204, assumes that all underlying security mechanisms are session-oriented. The current specification does not support security mechanisms, which encapsulate key distribution and other security context management information in a single message along with the data being protected (examples of such mechanisms include those accessed through the proposed internet IDUP-GSS-API interface). Changes to the SECIOP protocol would be required to support non-session-oriented protocols.

F.4 Mandatory Security Mechanisms

The current specification does not mandate any particular security mechanism which all secure ORBs must implement. This is because the submitters did not think it was possible to specify out-of-the-box interoperability adequately in the timescale of this submission.

F.5 Specific Security Policies

This specification includes some standard types of security policies for security functionality such as access control, audit, and security of invocations. These are aimed at general commercial users. Some enterprises may require other types of policies, for example, support of mandatory access controls. Where there is a sufficient market for such policies, new policies may be defined, providing they fit with the replaceability interfaces defined in this specification.

F.6 Other Audit Services

This specification only contains limited audit facilities, which allow audit records of security-relevant events to be collected. It does not include:

- Filtering of records after generation to further reduce the size of the audit trail.
- Routing audit records to a collection point for consolidation and analysis or routing some as alarms to security administrators. (However, routing may be done using the OMG Event Service, if that is secure enough.)
- Audit reporting or analysis tools to use the audit trails to track down problems.

F.7 Possible Enhancements

F.7.1 SECIOP Mechanism and Option Negotiation

This specification assumes the mechanism identifiers in the IOR allow the client to choose which mechanisms and options to use when communicating with this target. Therefore, it does not define protocol exchanges to allow the client and target to negotiate either mechanisms or options.

However, if the target supports a number of mechanisms and options, the size of the IOR could become larger than desirable. So in the future, it may be desirable to define protocol exchanges for mechanism negotiation, for example, using [19].

F.7.2 Further Key Distribution Options

The current CSI-ECMA protocol defines secret and public key options for key distribution and a hybrid option where secret keys are used within a domain, but public keys are used between domains. It does not define the protocol for use in the sort of hybrid system where the initiator uses secret key and target uses public key technology and vice versa.

This may be needed for interoperation between unlike domains. If so, further architectural options from ECMA 235 may need to be included in the specification.

F.7.3 Further Delegation Options at/above Level 2

The current level 2 specification supports restricting where an initiator's attributes can be used to targets identified by security name. Further options for restricting where a PAC may be delegated could be added (e.g., to restrict delegation to a particular delegation policy domain). This would require definition of further "qualifier attributes" in the CSI-ECMA protocol (see application trust groups in ECMA 235). It would also require administration of this, which would best be done by extending the security policy administration in "Administrator's Interfaces" on page 15-130.

Composite delegation of the initiator plus immediate invoker kind is described in the CSI protocol, but is not mandatory at level 2. Further composite delegation options, including traced delegation, could be added.

F.8 Interoperability when using Non-Repudiation

The optional Non-repudiation service in the CORBA Security specification generates NR tokens. This specification does not specify the technology used to generate these tokens or a standard form for them. Interoperability of evidence tokens would require a standard specification for such tokens.

This CSI specification is focused on inter-ORB interoperability, and therefore the IOR and SECIOP protocol as well. It does not specify the format of evidence tokens as they do not affect the SECIOP protocol. However, these evidence tokens may be passed between ORBs as parameters, and will not be understood by an ORB which does not use the same security technology.

In the future, a mandatory interoperability evidence token format should be defined, at least for a limited number of types of evidence. This is expected to be compatible with the public key mechanism specified in this specification and to use X.509 version 3 certificates.

F.9 Audit Trail Interoperability

The CORBA Security specification includes an Audit Channel interface which allows applications and ORBs to write records to the audit trail. The way this Audit Service routes the audit records is not defined. This could be done using the OMG Event Service or other means. Also, the stored/on-the-wire format of audit records is not defined.

So there is no standard OMG-defined method of bringing together audit records from different Audit Services.

F.10 Management

This specification contains only the management interfaces which are essential for security policy management. It specifies how to obtain and use security policy objects. However, it does not contain:

- Specification of facilities for handling domains, or policies other than those required for security policy administration.
- Specification of facilities for the management of some aspects of security. For example, it does not specify how to create and install permanent keys, as this is implementation-specific.

F.11 Reference Restriction

This specification requires the movement of credentials to delegate access rights from one object to another. Another technique of access rights delegation restricts the use of an object reference according to a set of criteria. This approach, known as reference restriction, is under study by a number of vendors, but is not ready for standardization at this time. The criteria used to restrict references could include:

- Whether an object has the right to assert certain privileges, such as act on behalf of a principal, act on behalf of a group of principals, act in a particular role, act with a particular clearance, etc.
- Whether the object reference has been limited to use within a given time interval.
- Whether a particular method can be used by an object holding the object reference.

Various techniques for restricting object references have been developed. Some use cryptographic methods, while others store state in the object associated with the restricted reference, allowing the object to decide if a method request meets the restricted reference use criteria.

It is anticipated that vendors will explore this type of access rights delegation and move towards the standardization of an interface supporting it in a submission to a future RFP.

F.12 Target Control of Message Protection

In the current specification, message protection can be specified by policy administration at both the client and the target object.

Requesting an operation on an object may result in many other objects being invoked. The CORBA security specification in this specification allows an intermediate object in such a chain of objects to delegate received credentials to the next object in the chain (subject to policy). However, the current specification does not allow the application to control when and where these credentials are used. A later specification may provide such controls to ride the default quality of protection selectively. Therefore, it could cause some messages to have different qualities of protection during a security association.

The target has no equivalent interface to request the quality of protection for a particular response. There are cases where this could be useful.

A future security specification should consider adding control of quality of protection by the target for individual responses.

F.13 Advanced Delegation Features

Requesting an operation on an object may result in many other objects being invoked. The CORBA security specification in this specification allows an intermediate object in such a chain of objects to delegate received credentials to the next object in the chain (subject to policy). However, the current specification does not allow the application to control when and where these credentials are used. A later specification may provide such controls.

If so, it is expected that a **set_controls** operation on the Credentials object will be added to enable the application to set the controls, and a matching **get_controls** operation to enable it to see what controls apply (see the **set_privileges** and **get_attributes** operations defined in Interfaces under Section 15.5.4, “The Credentials Object,” on page 15-94).

The **set_controls** operation would allow the application to specify a set of required control values such as delegation mode (allowing for richer forms of delegation), restrictions on where the credentials may be used and/or delegated, and validity period.

Note – These operations were not included in the specification because of concerns about portability of applications using them. Current delegation implementations use a wide variety of delegation controls, and some use similar controls in semantically different ways. Further implementation experience and investigation may make it possible to define a portable, standard set.

F.14 Overlapping and Hierarchical Domains

This specification does not require support for overlapping or hierarchical security policy domains. However, it is possible to implement both using the interfaces provided.

Recall from Section 15.6, "Administrator's Interfaces," on page 15-130, that the **DomainAccessPolicy** for each domain defines which rights are *granted* to subjects when they attempt to access objects in the domain. In order to make an access decision, the **AccessDecision** logic also needs to know which rights are *required* to execute the operations of an object, which is a member of the relevant domain. The **RequiredRights** interface provides this information; the **AccessDecision** object will probably use this interface in most implementations.

A **RequiredRights** instance can be queried to determine which rights a user must be granted in order to be allowed to invoke an object's operations. The intended use of **DomainAccessPolicy** and **RequiredRights** objects by the **AccessDecision** object is illustrated next, in Figure 15-68.

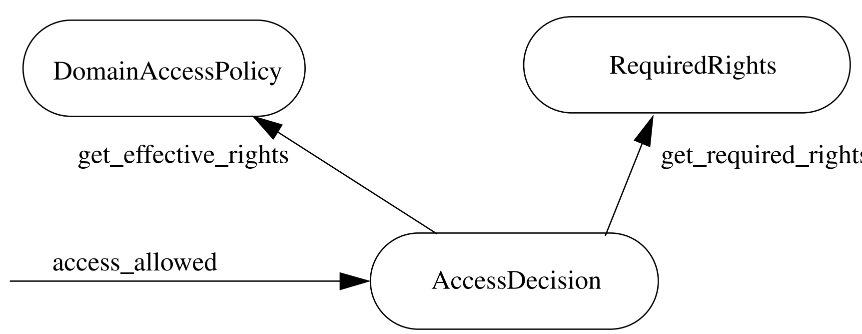


Figure 15-68 Intended Use by AccessDecision

AccessDecision retrieves the relevant policy object by calling **get_domain_managers** on the target object reference, and then calling **get_domain_policy(access)** on the returned domain manager (assuming for purposes of this example that there is only one). It then calls **get_effective_rights** on the returned policy object. **AccessDecision** then calls **get_required_rights** on **RequiredRights** and compares the returned list of required rights with the effective rights. If all required rights have been granted, it grants the access.

Figure 15-69 illustrates how the specification could be implemented to support overlapping access policy domains (i.e., to allow an object to be a member of more than one domain, such that each domain has an access policy and all domains' access policies are applied). In the diagram, the **AccessDecision** object must have logic to combine the policies asserted by the various AccessPolicy objects (which may involve evaluating which AccessPolicy object's policy takes precedence over the others). Note that the **AccessDecision** object knows the target object reference, because it is passed as an input parameter to the **access_allowed** operation.

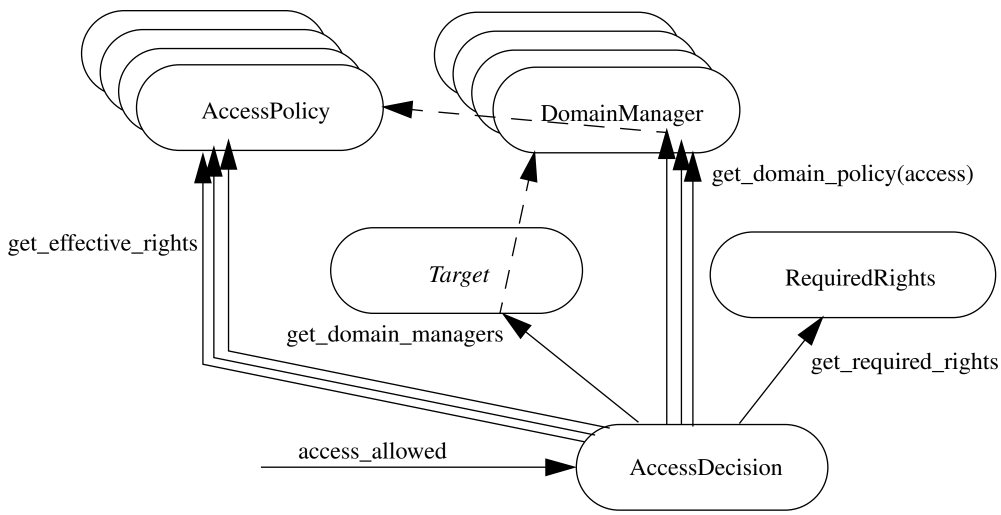


Figure 15-69 Supporting Overlapping Access Policy Domains

Hierarchical domains can be handled in a similar way as illustrated in Figure 15-70 on page 15-365 (note that once again the **AccessDecision** object's implementation is responsible for reconciling the various retrieved policies).

.

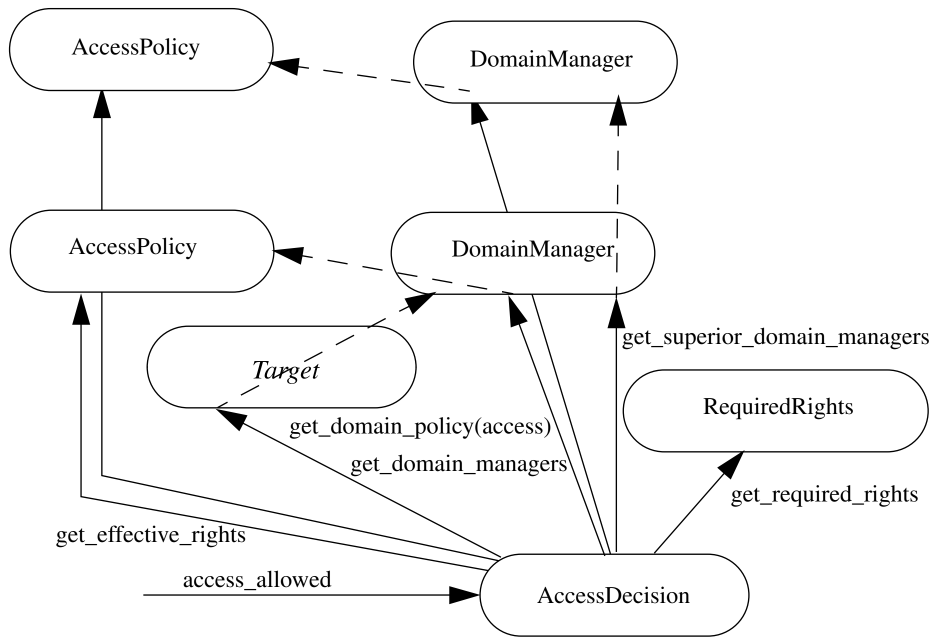


Figure 15-70 Hierarchical domains

F.15 Capability-Based Access Control

Capability-based systems store access policy information in tokens, which are passed from sender to receiver along with a message, rather than in tables associated with target objects or domains. In such systems, the **DomainAccessPolicy** object will generally not be used in resolving target-side access control checks. Instead, a CapabilityAccessPolicy object might be returned from a call to **Object::get_policy** in a capability-based system. This object could retrieve the granted rights from the capability (which will be associated with the requester's credentials), illustrated in Figure 15-71 on page 15-366.

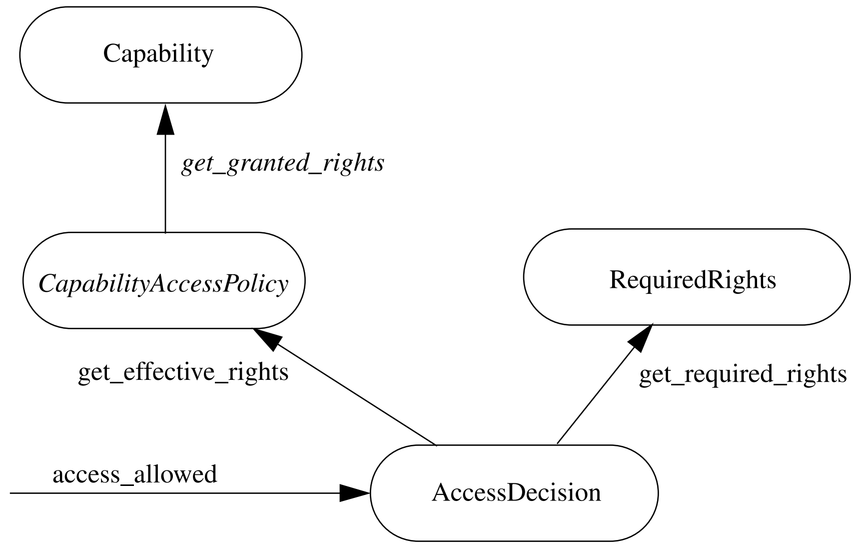


Figure 15-71 Retrieving Granted Rights

Note – Neither the CapabilityAccessPolicy interfaces nor the Capability interfaces are defined in this specification (the **get_granted_rights** call to the capability in Figure 15-71 is printed in italics, to indicate that no IDL is provided for it in this specification). The diagram assumes that CapabilityAccessPolicy inherits the **get_effective_rights** operation from **AccessPolicy**.

F.16 Non-repudiation Services

This specification contains Non-repudiation Services for evidence handling. It is anticipated that future service offerings could include data protection processing and the specification of a delivery service. In addition, it is expected that policy processing interfaces will emerge to cover the broad range of non-repudiation policy coverage within the service.

It is anticipated that the data protection and delivery service functions will be reaching a level of maturity within other standards domains (such as IETF and ISO SC27), which should allow a richer definition of these services to be enabled in future revisions of this specification.

The absence of these services in this specification means that application writers and manipulators will need to consult local implementation practice for the correct course of action to be taken when writing or porting their software.

This specification also does not include a standard format of evidence token for interoperability. In the future, a token format based on public key certificates may be specified.

Appendix G Interoperability Guidelines

G.1 Introduction

This appendix includes:

- Guidelines for defining Security Mechanism TAGs in Interoperable Object References (IORs).
- Examples of the secure inter-ORB protocol, SECIOP.

G.2 Guidelines for Mechanism TAG Definition in IORs

Section 15.8, "Security Interoperability Protocols," on page 15-172, defined a prototype TAG definition for security association mechanisms. This appendix provides guidelines that specifiers of mechanism TAGs (called authors here) should follow.

In addition to registering TAGs with the OMG, authors must lodge a document that explains how the mechanism (and its associated options) is mapped to this standard. Its document should:

- Identify the "security mechanism tagged component" being described. It may be either:
  - A new component TAG for the mechanism with a set of options it can have (for example, a separate TAG for each combination of mechanism and algorithm), or
  - Use TAG_GENERIC_SEC_MECH and specify the mechanism OID (for use in the **security_mechanism_type** field) being described by this specification.

  It may not be both.
- Specify the scope implied by the above mechanism identifier. This should not exceed:
  - Security association mechanism
  - Negotiation protocols
  - Cryptographic algorithms
  - Authentication method (e.g., public key)
- For the first example under the first bullet, describe the format, contents, and encoding of the **component_data** field for the TAG-specific components. For the second example under the first bullet, describe the format, contents, and encoding of the data in the **mech_specific_data** and components fields of the TAG-specific components. In each case, this may include:
  - Allocating new component TAGs and describing the format, contents, and encoding of their data.
  - Specifying the use of these new tagged components, as well as other predefined tagged components within TAG-specific components.
  - Specifying the use of these new tagged components, as well as other predefined tagged components that may or should appear at the top level of the multicomponent profile.

- Describe a model that should be followed when defining future extensions or variations using the same mechanism.
- The author must define either by reference to another document, or explicitly, the format of the context tokens used by the mechanism in the SECIOP protocol.

G.3 SECIOP Examples

G.3.1 Mutual Authentication

In this example, the client wishes to authenticate the identity of the target (in addition to the targets requirement to authenticate the client) before it is prepared to send a request to the target.

The client sends an **EstablishContext** message to the target containing the client's context id for the association, and the token required by the target to authenticate it and define the options chosen by the client for the association. The target verifies the client's token and generates the token required by the client to authenticate the target. The target sends this token (along with the client's context id for the association and its own) to the client in a **CompleteEstablishContext** message. When the client receives this message, it authenticates the target using the token supplied by the target and establishes the peer id as part of the context.

Having completed the establishment of the context, the client sends the request as part of a MessageInContext message, which includes the target's context identifier and the integrity token for the message. When the target receives the message, it identifies the context by its identifier, checks the integrity of the message with the token, and passes the message to GIOP. When the reply is returned, it is sealed for integrity and returned to the client in an SECIOP MessageInContext with the client identifier for the context and the generated integrity token.

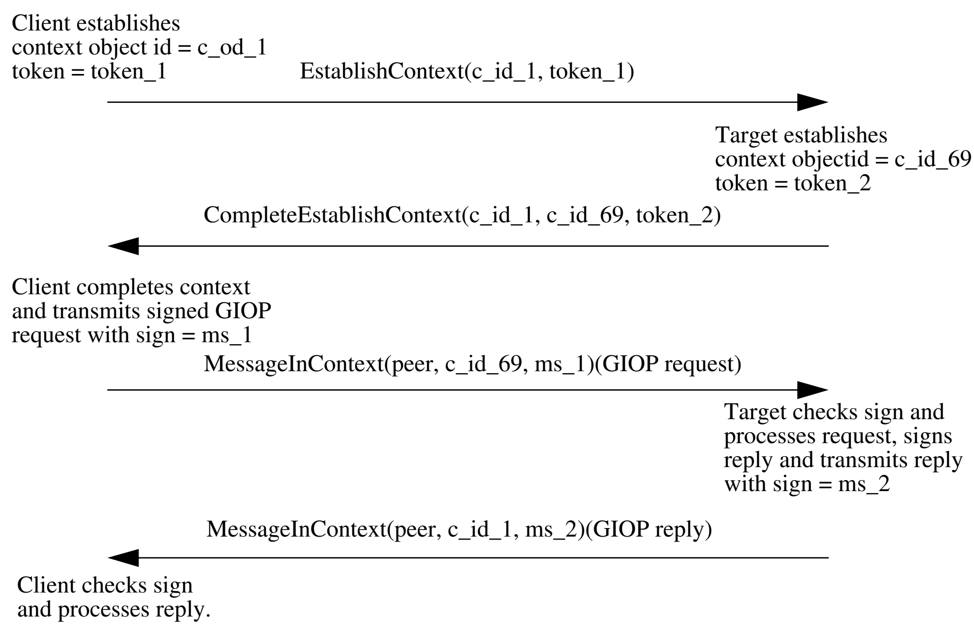


Figure 15-72 Mutual Authentication

G.3.2 Confidential Message with Context Establishment

This example describes how context establishment is combined with the transmission of a confidentiality-protected message when the client does not wish to authenticate the target before passing it a message.

The client establishes its context object with identifier c_id_1. This identifier is included with the token (token_1) in an **EstablishContext** message. The GIOP request is transformed into the message seal (ms_1) and sent with the client's context identifier in a **MessageInContext**.

When the target receives the message, it first processes the **EstablishContext** message, authenticating the client and allowing the target to create its context object. It then unseals the message in ms_1 and passes it to GIOP.

When GIOP sends the reply, SECIOP adds a **CompleteEstablishContext** message to the **MessageInContext** message, which protects the reply, to enable the target to return its context identifier to the client. When the client receives the message, it first completes its view of the context (adding the target's id to the state for the context). It can then unseal the reply from ms_2 and passes the reply message up the protocol stack.

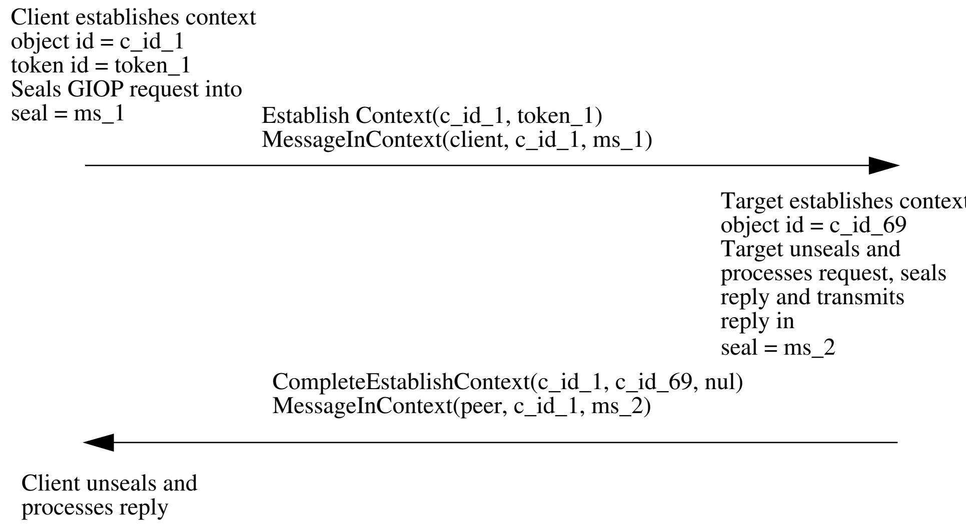


Figure 15-73 Confidential Message with Context Establishment

G.3.3 Fragmented GIOP Request with Context Establishment

In this example, the security context is established as part of the processing of a fragmented GIOP request (note that the current GIOP protocol does not support fragmentation, but this example indicates the independence of SECIOP from the current GIOP protocol and explains how the SECIOP protocol would handle a fragmented GIOP request). The sequence described reflects the requirement of the target to authenticate the client's privileges.

The client establishes its context object (with id c_id_1) and passes this identifier with the authentication token in an **EstablishContext** message. As the client does not require authenticating the target, this message is sent with a **MessageInContext** message with the integrity sign (ms_1) and the GIOP fragment (as the message field of the **MessageInContext**).

When the target receives the messages, it authenticates the client using token_1. It then creates a context object with c_id_69, and then processes the **MessageInContext**, checking the integrity of the message using sign ms_1. Having checked the message, it passes the fragment up the protocol stack.

The client sends the final fragment as a **MessageInContext** with sign ms_2, but as the target has not yet passed its identifier for the context to the client, the client uses its own identifier for the context.

The target finds its context object from the client's identifier (c_id_1) and checks the integrity of the message. It then passes the final fragment up the protocol stack to GIOP.

GIOP now has a complete request and can invoke the object (subject to the access decision function).

GIOP generates a single fragment reply, which is passed to the SECIOP protocol machine. The reply is sent within a **MessageInContext** with sign ms_3. In addition, a **CompleteEstablishContext** message is generated to allow the target to pass its identifier for the context (c_id_69) to the client for use in future messages.

The client receives the message and updates its context object to record the target's context identifier. It then checks the integrity of the **MessageInContext** and passes the reply up the protocol stack (to GIOP).

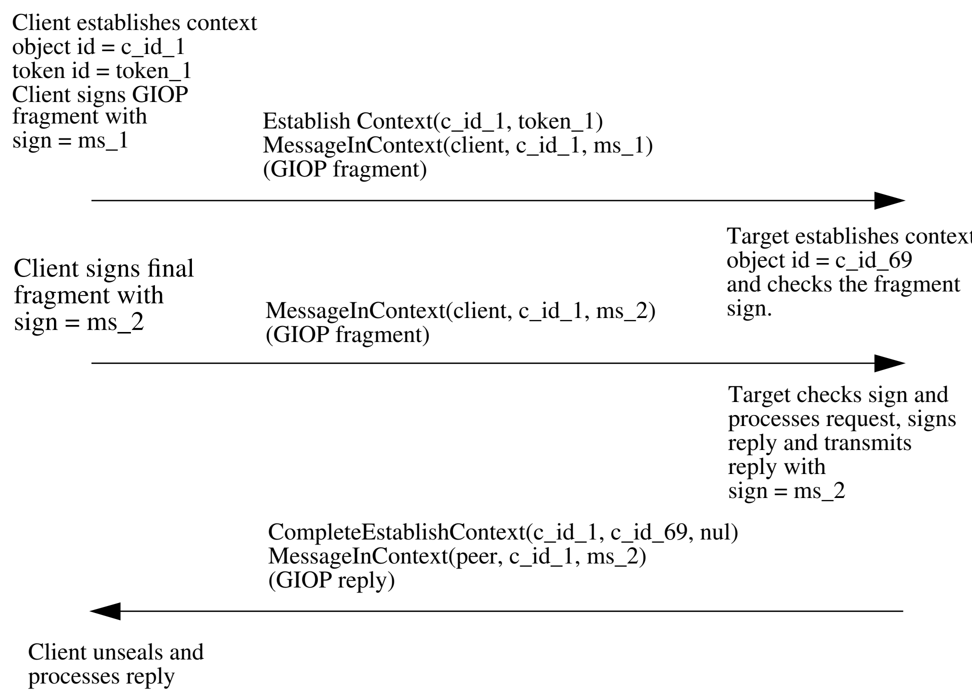


Figure 15-74 Fragmented GIOP Request with Context Establishment

Appendix H Glossary

H.1 Definitions

absolute time	Time relative to the time base of 0 hours 0 minutes 0 seconds of 15 October 1582 (c.f. CORBA Time Service [3]), accurate within a known margin of error.
access control	The restriction of access to resources to prevent its unauthorized use.
access control information (ACI)	Information about the initiator of a resource access request, used to make an access control enforcement decision.
access conrol list	A list of entities, together with their access rights, which are authorized to have access to a resource.
access decision function	The function which is evaluated in order to make an access control enforcement decision. The inputs to an access decision function include the requester's access control information (q.v.), the resource's control information, and context data.
access decision object (ADO)	The CORBA security object which implements access decision functions.
accountability	The property that ensures that the action of an entity may be traced uniquely to the entity.
active threat	The threat of a deliberate unauthorized change to the state of a system.
adjudicator	An authority that resolves disputes among parties in accordance with a policy. In CORBA security, an adjudicator evaluates non-repudiation evidence in order to resolve disputes.
anonymous user	A user of the system operating under a distinguished "public" identity corresponding to no specific user.
assurance	1. Justified confidence in the security of a system. 2. Development, documentation, testing, procedural, and operational activities carried out to ensure that a system's security services do in fact provide the claimed level of protection.

asymmetric key	One half of a key pair used in an asymmetric ("public-key") encryption system. Asymmetric encryption systems have two important properties: (i) the key used for encryption is different from the one used for decryption (ii) neither key can feasibly be derived from the other.
audit	See security audit.
audit event	The data collected about a system event for inclusion in the system audit log.
audit trail	See security audit trail.
authentication	The verification of a claimant's entitlement to use a claimed identity and/or privilege set.
authentication information	Information used to establish a claimant's entitlement to a claimed identity (a common example of authentication information is a password).
authorization	The granting of authority, which includes the granting of access based on access rights.
availability	The property of being accessible and usable upon demand by an authorized user.
call chain	The series of client to target object calls required to complete an operation. Used in this specification in conjunction with delegation.
certification authority	A party trusted to vouch for the binding between names or identities and public keys. In some systems, certification authorities generate public keys.
ciphertext	The result of applying encryption to input data; encrypted text.
cleartext	Intelligible data; text which has not been encrypted or which has been decrypted using the correct key. Also known as "plaintext."
confidentiality	The property that information is not made available or disclosed to unauthorized individuals, entities, or processes.
conformance level	A graduated sequence of defined sets of functionality defined by the CORBA Security specification. An implementation must implement at least one of these defined sets of functionality in order to claim conformance to CORBA Security.

conformance option	A defined set of functionality which implementations may optionally provide in order to claim CORBA Security-conformant functionality over and above the minimum required by the defined conformance levels.
cofnormance statement	A written document describing the conformance levels and conformance options to which an implementation of the OMG CORBA Security specification conforms.
control attributes	The set of characteristics which restrict when and where privileges can be invoked or delegated.
counter-measures	Action taken in response to perceived threats.
credentials	Information describing the security attributes (identity and/or privileges) of a user or other principal. Credentials are claimed through authentication or delegation (q.v.) and used by access control (q.v.).
current object	An object representing the current execution context; CORBA Security associates security state information, including the credentials of the active principal, with the current object.
Discretionary Access Control (DAC)	An access control policy regime wherein the creator of a resource is permitted to manage its access control policy information.
data integrity	The property that data has not been undetectably altered or destroyed in an unauthorized manner or by unauthorized users.
DCE	Distributed Computing Environment (of OSF).
DCE CIOP	DCE Common Inter-ORB Protocol - the protocol specified in the OMG CORBA 2.0/ Interoperability specification which uses the DCE RPC for interoperability.
decipherment	Generation of cleartext from ciphertext by application of a cryptographic algorithm with the correct key.
decryption	See decipherment.
delegation	The act whereby one user or principal authorizes another to use his (or her or its) identity or privileges, perhaps with restrictions.
denial of service	The prevention of authorized access to resources or the delaying of time-critical operations.

digital signature	Data appended to, or a cryptographic transformation of, a data unit that allows a recipient of the data unit to prove the source and integrity of the data against forgery, e.g., by the recipient.
domain	A set of objects sharing a common characteristic or abiding by a common set of rules. CORBA Security defines several types of domains, including security policy domains, security environment domains, and security technology domains.
domain manager	A CORBA Security object through whose interfaces the characteristics of a security policy domain are administered.
encipherment	Generation of ciphertext from corresponding cleartext by application of a cryptographic algorithm and a key.
encryption	See encipherment.
ESIOP	See encipherment.**evidence**: Data generated by the CORBA Security Non-Repudiation service to prove that a specific principal initiated a specific action.
evidence token	A data structure containing CORBA Security Non-Repudiation evidence.
federated domains	Separate domains whose policy authorities have agreed to a set of shared policies governing access by users from one domain to resources in another.
GSS-API	Generic Security Services- Application Programming Interface - specified by RFC 1508 issued by the Internet IETF. An update to this interface is near completion as this is written, and it is anticipated that RFC 1508 will be superseded by a revised specification soon.
GIOP	General Inter-ORB Protocol (specified in the OMG CORBA 2.0/ Interoperability specification).
group	A CORBA Security privilege attribute. Many users (and other principals) may be assigned the same group attribute; this allows administrators to simplify security administration by granting rights to groups rather than to individual principals.
granularity	The relative fineness or coarseness by which a mechanism may be adjusted.
hierarchical domains	A set of domains together with a precedence hierarchy defining the relationships among their policies.

identity	A security attribute with the property of uniqueness; no two principals' identities may be identical. Principals may have several different kinds of identities, each unique (for example, a principal may have both a unique audit identity and a unique access identity). Other security attributes (e.g., groups, roles, etc.) need not be unique.
immediate invoker	In a delegated call chain, the client from which an object directly receives a call.
impersonation	The act whereby one principal assumes the identity and privileges of another principal without restrictions and without any visible indication to recipients of the impersonator's calls that delegation has taken place.
initiator	The first principal in a delegation "call chain;" the only participant in the call chain which is not the recipient of a call.
integrity	In security terms, the property that a system always faithfully and effectively enforces all of its stated security policies.
interceptor	An object which provides one or more specialized services at the ORB invocation boundary, based upon the context of the object request. The OMG CORBAsecurity specification defines the security interceptors.
intermediate	An object in a delegation "call chain" which is neither the initiator nor the ultimate (final) target.
IETF	Internet Engineering Task Force. Reviews and issues Internet standards.
IIOP	Internet Interoperable Object Protocol (specified in the OMG CORBA 2.0/ Interoperability specification).
IOR	Interoperable Object Reference - a data structure specified in the OMG CORBA 2.0/ Interoperability specification.
ITSEC	Information Technology Security Evaluation Criteria (of ECSC-EEC-EAEC). Harmonized Criteria.
Mandatory Access Control (MAC)	An access control regime wherein resource access control policy information is always managed by a designated authority, regardless of who creates the resources.

locality-constrained	An object is locality-constrained if it cannot be accessed from outside a specific locality. references to the object cannot be meaningfully passed outside the boundaries of the locality of concern.
mechanism	A specific implementation of security services, using particular algorithms, data structures, and protocols.
message protection	Security protection applied to a message to protect it against unauthorized access or modification in transit between a client and a target.
mutual authentication	The process whereby each of two communicating principals authenticates the other's identity. Frequently this is a prerequisite for the establishment of a secure association between a client and a target.
Non-repudiation	The provision of evidence which will prevent a participant in an action from convincingly denying his responsibility for the action.
ORB Core	The functionality provided by the CORBA Object Request Broker which provides the basic representations of objects and the communication of requests.
ORB Services	Elements of functionality provided transparently to applications by the CORBA Object Request Broker in response to the implicit context of an object request.
ORB technology domain	A set of objects or entities that share a common ORB implementation technology.
originator	The entity in an object request which creates the request.
passive threat	The threat of unauthorized disclosure of information without changing the state of the system.
physical security	The measures used to provide physical protection of resources against deliberate and accidental threats.
POSIX	Portable Open System Interfaces (for) UNIX - A set of standardized interfaces to UNIX systems specified by IEEE Standard 1003.
principal	A user or programmatic entity with the ability to use the resources of a system.
privacy	1. See confidentiality. 2. The right of individuals to control or influence what information related to them may be collected and stored and by whom that information may be disclosed.

private key	In a public-key (asymmetric) cryptosystem, the component of a key pair which is not divulged by its owner.
privilege	A security attribute (q.v.) which need not have the property of uniqueness, and which thus may be shared by many users and other principals. Examples of privileges include groups, roles, and clearances.
proof of delivery	Non-repudiation evidence demonstrating that a message or data has been delivered.
proof of origin	Non-repudiation evidence identifying the originator of a message or data.
proof of receipt	Non-repudiation evidence demonstrating that a message or data has been received by a particular party.
protection boundary	The domain boundary within which security services provide a known level of protection against threats.
Protocol Data Unit (PDU)	The data fields of a protocol message, as distinguished from the protocol header and trailer fields.
POA	Portable Object Adaptor
proof of submission	Non-repudiation evidence demonstrating that a message or data has been submitted to a particular principal or service.
public key	In a public-key (asymmetric) cryptosystem, the component of a key pair which is revealed.
public-key cryptosystem	An encryption system which uses an asymmetric-key (q.v.) cryptographic algorithm.
Quality of Protection (QOP)	The type and strength of protection provided by a message-protection service.
RPC	Remote Procedure Call
replaceability	The quality of an implementation which permits substitution of one security service for another semantically similar service.
repudiation	Denial by one of the entities involved in an action of having participated in all or part of the action.
RFP	Request for Proposal. An OMG procedure for soliciting technology from OMG members.

right	A named value conferring the ability to perform actions in a system. Access control policies grant rights to principals (on the basis of their security attributes); in order to make an access control decision, access decision functions compare the rights granted to a principal against the rights required to perform an operation.
rights type	A defined set of rights.
role	A privilege attribute representing the position or function a user represents in seeking security authentication. A given human being may play multiple roles and therefore require multiple role privilege attributes.
RSA	An asymmetric encryption algorithm invented by Ron Rivest, Adi Shamir, and Len Adelman.
seal	To encrypt data for the purpose of providing confidentiality protection.
secret-key cryptosystem	A cryptosystem which uses a symmetric-key (q.v.) cryptographic algorithm.
secure time	A reliable Time service that has not been compromised, and whose messages can be authenticated by their recipients.
security association	The shared security state information which permits secure communication between two entities.
security attributes	Characteristics of a subject (user or principal) which form the basis of the system's policies governing that subject.
security audit	The facility of a secure system which records information about security-relevant events in a tamper-resistant log. Often used to facilitate an independent review and examination of system records and activities in order to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, to detect breaches in security, and to recommend changes in control, policy, and procedures.
security features	Operational information which controls the security protection applied to requests and responses in a CORBA Security-conformant system.
security context	The CORBA Security object which encapsulates the shared state information representing a security association.

security policy	The data which defines what protection a system's security services must provide. There are many kinds of security policy, including access control policy, audit policy, message protection policy, non-repudiation policy, etc.
security policy domain	A domain whose objects are all governed by the same security policy. There are several types of security policy domain, including access control policy domains and audit policy domains.
security service	Code that implements a defined set of security functionality. Security services include Access Control, Audit, Non-repudiation, and others.
security technology domain	A set of objects or entities whose security services are all implemented using the same technology.
subject	An active entity in the system; either a human user principal or a programmatic principal.
symmetric key	The key used in a symmetric ("secret-key") encryption system. In such systems, the same key is used for encryption and decryption.
tagged profile	The data element in an IOR which provides the profile information for each protocol supported.
target	The final recipient in a delegation "call chain." The only participant in such a call chain which is not the originator of a call.
target ACI	The Access Control Information for the target object.
target object	The recipient of a CORBA request message.
threat	A potential violation of security.
traced delegation	Delegation wherein information about the initiator and all intervening intermediates is available to each recipient in the call chain, or to the authorization subsystem controlling access to each recipient.
trust model	A description of which components of the system and which entities outside the system must be trusted, and what they must be trusted for, if the system is to remain secure.
trusted code	Code assumed to always perform some specified set of operations correctly.

trusted computing base (TCB)	The portion of a system which must function correctly in order for the system to remain secure. A TCB should be tamper-proof and its enforcement of policy should be noncircumventable. Ideally, a system's TCB should also be as small as possible, to facilitate analysis of its integrity.
TCSEC	Trusted Computer System Evaluation Criteria (a U.S. Department of Defense Standard specifying requirements for secure systems).
unauthorized principal	A user or other principal who has not authenticated any identity or privilege.
UNO	Universal Networked Objects (an OMG Specification, now obsolete).
UTC	Coordinated Universal Time
unsecure time	Time obtained from an unsecure time service.
UTO	Universal Time Object (c.f., CORBA Time Service [3])
user	A human being using the system to issue requests to objects in order to get them to perform functions in the system on his behalf.
user sponsor	The interactive user interface to the system which acts as the authenticating authority (e.g., validating passwords) which validate the identity of a user.
vault	The CORBA Security object which creates security context objects.
X/Open	X/Open Company Ltd., U.K.

H.2 References

The following sources were used in the preparation of this glossary:

Applied Cryptography, 2nd edition by Bruce Schneier, John Wiley and Sons, New York, 1996.

ISO Standard 7498-2, “Information Processing Systems -- Open Systems Interconnection -- Basic Reference Model -- Part 2:Security Architecture”, International Standards Organization,1989.

ECMA TR/46 “Security in Open Systems: A Security Framework”, European Computer Manufacturers Association, 1988.

ITSEC “Information Technology Security Evaluation Criteria” European Commission, 1991.

DoD Standard 5200.28-STD “Department of Defense Trusted Computer System Evaluation Criteria”, US Department of Defense, 1985.

X/Open Snapshot: “Distributed Security Framework: Company Review Draft”, X/Open Company Ltd.,U.K. 1994.

Computer Related Risks: Peter G. Neuman, The ACM Press, 1995

Appendix I References

Note that these references are to definitions which are sometimes a set of documents.

[1] CORBA/IIOP 2.2.

[2] Common Secure IIOP Request for Proposals (orb/96-01-03)

[3] CORBA Time Service, Chapter 16 of CORBAservices specification, also available at the URL http://www.omg.org/docs/formal/97-02-22.pdf

[4] IETF RFC 1779 A String Representation of Distinguished Names. March 1995.

[5] *X/Open Application Environment Specification for Distributed Computing*.

[6] X/Open Preliminary Specification *X/Open DCE: Authentication and Security Services.*

[7] X/OPEN *CAE Specification C309*

[8] OSF *AES/Distributed Computing RPC Volume*.

[9] *OSF DCE 1.1 Application Development Reference*

[10] The ECMA GSS-API mechanism specified in ECMA-235. See also related standard ECMA-219 (Authentication and Privilege Attribute Security Application with related key distribution functions)

[11] GSS-APIThe Generic Security Services API as defined in IETF RFC 1508 (September 1993) and X/Open P308.An update to RFC 1508 has been produced by the IETF cat group.

[12] The IETF GSS Kerberos V5 definition which specifies details of the use of Kerberos V5 with GSS-API. It includes updates to RFC 1510 e.g. how to carry delegation information. It is specified in RFC 1964.

[13] The Kerberos V5 mechanism as defined in IETF RFC 1510 (September 1993).

[14] The ORB Portability Specification - CORBA V2.3 Chapter 9..

[15] Open Distributed Processing - Reference Model Parts 1 through 3, OMG doc #om/96-10-02, 03, 04.

[16] The SESAME gss-api mechanism. This is a subset of the ECMA GSS Mechanism and is specified in draft-ietf-cat-sesamemech-00.txt.

[17] The SESAME V4 Overview. This can be found via the web at www.esat.kuleuven.ac.be/cosic/sesame.html

[18] *John G. Fletcher, "Serial Link Protocol Design: A Critique of the X.25* Standard, Level 2," Proceedings of SIGCOMM '84, ACM SIGCOMM, pp.26-33, June 6-8, 1984.

[19] Simple negotiation GSS-API mechanism as defined in draft-ietf-cat-snego-02.txt.

[20] The Simple Public-Key GSS-API Mechanism (SPKM). Internet Draft draft-ietf-cat-spkmgss-06.txt Jan. 1996.

[21] Secure Socket Layer [ftp://ierf.cnsi.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt]

[22] ISO/IEC 9594-8, “Information Technology - Open Systems Interconnection - The Directory: Authentication Framework”, CCITT/ITU Recommendation X.509, 1993.

[23] The extended gss-api supporting access control and delegation extensions defined in draft-ietf-cat-xgssapi-acc-cntrl-00.txt. This interface is also defined in the ECMA GSS-API Mechanism standard - ECMA-235

Trading Object Service Specification 16

This chapter provides complete documentation for the Trading Object Service specification.

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	16-2
“Concepts and Data Types”	16-4
“Exceptions”	16-23
“Abstract Interfaces”	16-28
“Functional Interfaces”	16-30
“Service Type Repository”	16-59
“Dynamic Property Evaluation interface”	16-67
“Conformance Criteria”	16-68
Appendix A, “CORBA OMG IDL based Specification of the Trading Function”	16-74
Appendix B, “OMG Constraint Language BNF”	16-93
Appendix C, “OMG Constraint Recipe Language”	16-99

16.1 Overview

The OMG trading object service facilitates the offering and the discovery of instances of services of particular types. A trader is an object that supports the trading object service in a distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities. Advertising a capability or offering a service is called "export." Matching against needs or discovering services is called "import." Export and import facilitate dynamic discovery of, and late binding to, services.

To export, an object gives the trader a description of a service and the location of an interface where that service is available. To import, an object asks the trader for a service having certain characteristics. The trader checks against the service descriptions it holds and responds to the importer with the location of the selected service's interface. The importer is then able to interact with the service. These interactions are shown in Figure 16-1.

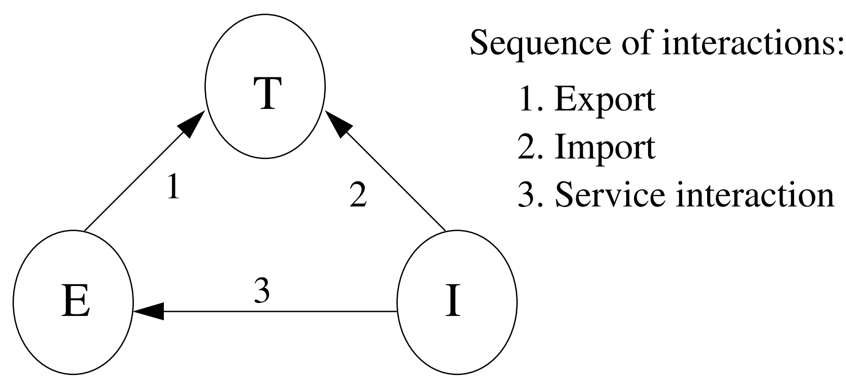


Figure 16-1 Interactions between a trader and its clients

Due to the number of service offers that will be offered worldwide, and the differing requirements that users of a trading service will have, it is inevitable that a trading service will be split up and the service offers will be partitioned.

Each partition will, in the first instance, meet the trading needs of a community of clients (exporters and importers). Where a client needs a scope for its trading activities that is wider than that provided by one partition, it will access other partitions either directly or indirectly. Directly means that the client interacts with the traders handling those partitions. Indirectly means that the client interacts with one trader only and this trader interacts with other traders responsible for other partitions. The latter possibility is referred to as interworking (or federation) of traders.

The trading object service in an OMG environment allows interworking between traders and objects to:

- export (advertise) services
- import information about one or more exported services, according to some criteria

16.1.1 Diversity and Scalability

The concept of trading to discover new services applies to a wide range of scenarios. A trader may contain numerous offers of service and its implementation may be based upon a database. Or, a trader may contain only a few offers and be implementable as a memory resident trader. These two cases exhibit different qualities: availability and integrity in the first case and performance in the second. The variation in these scenarios illustrates the need for scalability, both upwards for very large systems and downwards for small, fast systems.

To discover any arbitrary offer of service, a trader needs all offers to be visible to it. One partition cannot hold every offer, many are held at other partitions; therefore, in addition to a number of offers, a trader must possess information about other partitions. However, there is no need for a trader to know about all other partitions. Some of this knowledge can be utilized indirectly via other traders.

The partitioning of the offer space and the limited knowledge held within one partition about other partitions is the basis for meeting requirements for both distribution and contextualisation of the trading object service.

16.1.2 Linking Traders

The requirements to contextualise the offer space and to distribute the trading object service are both met by linking traders together. When a trader links to other traders, it makes the offer spaces of those traders implicitly available to its own clients.

Each trader has a horizon limited to those other traders to which it is explicitly linked. As those traders are linked to yet more traders, a large number of traders are reachable from a given starting trader. The traders are linked to form a directed graph with the information describing the graph distributed among the traders. This graph is called the trading graph.

Links may cross domain boundaries (e.g., administrative, technological, etc.); therefore, trading is a federated system (i.e., one that spans many domains).

16.1.3 Policy

To meet the diverse requirements likely to be placed upon the trading function, some degree of freedom is necessary when specifying the behavior of a trader object. To accomplish this, and yet still meet the goals of this specification, the concept of policy is used to provide a framework for describing the behavior of any OMG trading object service implementation.

This specification identifies a number of policies and gives them semantics. Each policy partly determines the behavior of a trader.

Policies may be communicated during interaction, in which case they relate to an expectation on subsequent behavior.

16.1.4 Additional ObjectID

A trading object service may be used by an object to bootstrap itself into operation; as such, this specification mandates an additional ObjectId for use in the resolve_initial_references() operation defined in the ORB Initialization Specification, OMG Document 94-10-24.

The following ObjectId is reserved for finding an initial trading object service:

```
TradingService
```

As described in 94-10-24, a client object wishing to obtain an initial trading object service object reference will invoke the resolve_initial_references() operation, which has the following OMG IDL signature:

```
typedef string ObjectId;
exception InvalidName {};

Object resolve_initial_references (in ObjectId identifier) raises (InvalidName);
```

The object reference returned as the result of a successful invocation of this operation when “TradingService” is specified as the ObjectId parameter must be narrowed to an object reference of the appropriate type; for the trading object service this type is CosTrading::Lookup.

No other extensions are proposed to OMG IDL, CORBA, and/or the OMG object model.

16.2 Concepts and Data Types

16.2.1 Exporter

An exporter advertises a service with a trader. An exporter can be the service provider or it can advertise a service on behalf of another.

16.2.2 Importer

An importer uses a trader to search for services matching some criteria. An importer can be the potential service client or it can import a service on behalf of another.

16.2.3 Service Types

A service type, which represents the information needed to describe a service, is associated with each traded service. It comprises:

- an interface type which defines the computational signature of the service interface, and

- zero or more named property types. Typically these represent behavioral, non-functional, and non-computational aspects that are not captured by the computational signature.

The property type defines the property value type, whether a property is mandatory, and whether a property is readonly. That is, associated with a property type is the triple of <name, type, mode>, where the modes are:

```
enum PropertyMode {
    PROP_NORMAL, PROP_READONLY,
    PROP_MANDATORY, PROP_MANDATORY_READONLY
};
```

A service type repository is used to hold the type information.

```
typedef Object TypeRepository;
```

Each service type in a repository is identified by a unique ServiceTypeName.

```
typedef Istring ServiceTypeName; // similar to IR::Identifier
```

An exporter specifies the service type of the service it is advertising; an importer specifies the service type it is seeking.

Service types can be related in a hierarchy that reflects interface type inheritance and property type aggregation. This hierarchy provides the basis for deciding if a service of one type may be substituted for a service of another type. These considerations are described more fully in the following service type model.

Service Type Model

The service type model is illustrated by the following BNF:

```
service <ServiceTypeName>[:<BaseServiceTypeName>
[,<BaseServiceTypeName>]*]{
    interface <InterfaceTypeName>;
    [[mandatory] [readonly] property <IDLType> <PropertyName>;]*
};
```

The keyword “service” introduces a new ServiceTypeName. Its structure is similar to that of interface repository identifiers (::First::Second::Third ...). As the service type is visible to end users and not just to programmers, it is internationalizable.

The list of BaseServiceTypeNames lists those service types from which this service type is derived, which in turn defines where services of this service type can substitute for other service.

The “interface” keyword introduces the InterfaceTypeName for this service. It is related by equivalence or by derivation to the InterfaceTypeNames in each of the BaseServiceTypeNames.

The properties clause is a list of property declarations. Each property declaration is marked by the keyword "property" and may be preceded by mode attributes "mandatory" and/or "readonly." A property declaration is completed by an IDLType and a PropertyName. A service must support all the properties of each of its base service types, they must have identical property value types, and they must not lose any property mode attributes.

The property mode attributes have the following connotations:

- mandatory - an instance of this service type ***must*** provide an appropriate value for this property when exporting its service offer.
- readonly - if an instance of this service type provides an appropriate value for this property when exporting its service offer, the value for this property may not be changed by a subsequent invocation of the Register::modify() operation.

The property strength graph is shown in Figure 16-2.

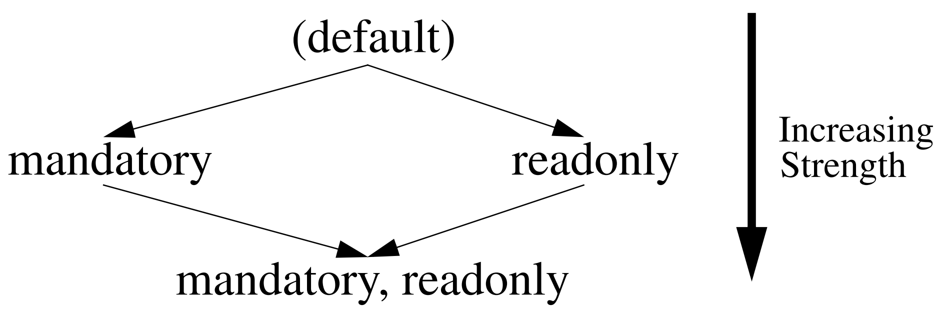


Figure 16-2 Property Strength

Summarizing, if a property is defined without any modifiers, it is optional (i.e., an offer of that service type is not required to provide a value for that property name, but if it does, it must be of the type specified in the service type), and the property value subsequently may be modified. The "mandatory" modifier indicates that a value must be provided, but that subsequently it may be modified. The "readonly" modifier indicates that the property is optional, but that once given a value, subsequently it may not be modified. Specifying both modifiers indicates that a value must be provided and that subsequently it may not be modified.

From the above discussion, one can state the rules for service type conformance; a service type β is a subtype of service type α, if and only if:

- the interface type associated with β is either the same as, or derived from, the interface type associated with α
- all the properties defined in α are also defined in β
- for all properties defined in both α and β, the mode of the property in β must be the same as, or stronger than, the mode of the property in α
- all properties defined in β that are also defined in α shall have the same property value type in β as their corresponding definitions had in α

16.2.4 Properties

Properties are <name, value> pairs. An exporter asserts values for properties of the service it is advertising. An importer can obtain these values about a service and constrain its search for appropriate offers based on the property values associated with such offers.

```
typedef Istring PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any PropertyValue;
struct Property {
    PropertyName name;
    PropertyValue value;
};
typedef sequence<Property> PropertySeq;

enum HowManyProps { none, some, all };
union SpecifiedProps switch ( HowManyProps) {
    case some: PropertyNameSeq prop_names;
};
```

16.2.5 Service Offers

A service offer is the information asserted by an exporter about the service it is advertising. It contains:

- the service type name,
- a reference to the interface that provides the service, and
- zero or more property values for the service.

An exporter must specify a value for all mandatory properties specified in the associated service type. In addition, an exporter can nominate values for named properties that are not specified in the service type. In such case, the trader is not obliged to do property type checking.

```
struct Offer {
    Object reference;
    PropertySeq properties;
};
typedef sequence<Offer> OfferSeq;

struct OfferInfo {
```

```
        Object reference;
        ServiceTypeName type;
        PropertySeq properties;
    };
```

Modifiable Properties

The value of a property in a service offer can be modified, if

- the property mode is not readonly, whether optional or mandatory, and
- the trader supports the modify property functionality.

Such property values can be updated by explicit modify operations to the trader. An exporter can control a service offer to be non-modifiable by exporting services with service types that have readonly properties. The modify operation will return a NotImplemented exception if a trader does not support the modify property functionality. An importer can also specify whether or not a trader should consider offers with modifiable properties during matching.

Dynamic Properties

A service offer can contain dynamic properties. The value for a dynamic property is not held within a trader, it is obtained on-demand from the interface of a dynamic property evaluator nominated by the exporter of the service. That is, a level of indirection is required to obtain the value for a dynamic property. The structure of a dynamic property value is:

```
    exception DPEvalFailure {
        CosTrading::PropertyName name;
        CORBA::TypeCode returned_type;
        any extra_info;
    };

    interface DynamicPropEval {

        any evalDP (
            in CosTrading::PropertyName name,
            in CORBA::TypeCode returned_type,
            in any extra_info
        ) raises (
            DPEvalFailure
        );
    };
```

```
struct DynamicProp {
    DynamicPropEval eval_if;
    CORBA::TypeCode returned_type;
    any extra_info;
};
```

It contains the interface to the dynamic property evaluator, the data type of the returned dynamic property, and any extra implementation dependent information. The trader recognizes this structure and, when the value of the property is required, invokes the evalDP operation from the appropriate DynamicPropEval interface. The dynamic property evaluator interface has only one operation, whose signature is defined in this standard for portability but its behavior is not specified. The only restrictions imposed are that the property must not be readonly and that the trader must support the dynamic property functionality.

The use of such Properties has implications on the performance of a trader. An importer can specify whether or not a trader should consider offers with dynamic properties during matching.

16.2.6 Offer Identifier

An offer identifier is returned to an exporter when a service offer is advertised in a trader. It identifies the exported service offer and is quoted by the exporter when withdrawing and modifying the offer (where supported). It only has meaning to the trader with which the service offer is registered.

```
typedef string OfferId;
typedef sequence<OfferId> OfferIdSeq;
```

16.2.7 Offer Selection

The total service offer space for an offer selection may be very large, including offers from all linked traders. Logically, the trader uses policies to identify the set S1 of service offers to examine. The service type and constraint is applied to S1 to produce the set S2 that satisfies the service type and constraint. Then this is ordered using preferences before returning the offers to the importer.

Standard Constraint Language

Importers use service type and a constraint to select the set of service offers in which they have an interest. The constraint is a well formed expression conforming to a constraint language.

This document defines the standard, mandatory language which is necessary for interworking between traders. Appendix B defines the syntax and the expressive power of the constraint language. This constraint language is used to write standard constraint expressions.

```
typedef Istring Constraint;
```

Its main features are:

Property Value Types	These manipulations are restricted to int, float, boolean, Istring/string, Ichar/char types, and sequences thereof. The character based types are ordered using the collating sequence in effect for the given character set. Types outside of this range can only be the subject of the "exist" operator.
Literals	In the constraint, literals are dynamically coerced as required for the properties they are working with. Literals can contain Istring.
Operators	The operators are comparison, boolean connective, set inclusion, substring, arithmetic operators, property existence.

Note – If a proprietary constraint language (outside the scope of this specification) is used, then the name and version of the constraint language is placed between << >> at the start of the constraint expression, The remainder of the string is not interpreted by a trader that does not support the quoted proprietary constraint language.

Preferences

Preferences are applied logically to the set of offers matched by application of the service type, constraint expression, and various policies. Application of the preferences can determine the order used to return matched offers to the importer.

```
typedef Istring Preference;
```

Consider the preference string as being composed of two portions.

- The first portion can be comprised of any of the following case-sensitive keywords:

  max min with random first

- The interpretation for the second portion is dependent on the first portion; it may be empty. Table 16-1 describes the preferences.

Table 16-1 Preferences

Preference	Description
max expression	The expression is numeric. The matched offers are returned in a descending order of the expression.
min expression	The expression is numeric. The matched offers are returned in an ascending order of the expression.
with expression	The expression is a constraint expression. The matched offers are ordered such that those that are TRUE precede those that are FALSE.
random	The order of returned matched offers is according to the following algorithm: select an offer at random from the set of matched offers, select another offer at random from the remaining set of matched offers, ..., select the single remaining offer.
first	The order of returned matched offers is in the order as the offers are discovered.

If no preference is specified, then the default preference of first applies. No combinations of the preferences are permitted.

The expression associated with max, min, and with can refer to properties associated with the matching offers. When applying a preference expression to the set of offers that match the service type and constraint expression, the offer set is partitioned into a group of offers for which the preference expression

- could be evaluated (ordered according to min, max, with), and
- could not be evaluated (e.g., the preference expression refers to a property name that is optional for that service type).

The offers are returned to the importer in the order of first group in their preference order, followed by those in the second group.

Note – If a proprietary preference language (outside the scope of this specification) is used, the name and version of the preference language used is placed between << >> at the start of the preference. The remainder of the string is not interpreted by a trader that does not support the quoted proprietary language.

Links

Links represent paths for propagation of queries from a source trader to a target trader. Each link corresponds to an edge in a trading graph, in which the vertices are traders. A link describes the knowledge that one trader has of another trading service that it uses. It also includes information of when to propagate or forward an operation to the target trader. A link has the following information associated with it:

- A Lookup interface provided by the target trader, which supports the query operation.
- A Register interface provided by the target trader, which supports the resolve operation.
- The link's default follow behavior, which may be used and is passed on when an importer does not specify a link_follow_rule policy.
- The link's limiting follow behavior, which overrides an importer's link_follow_rule if the importer's request exceeds the limit set by the link.

```
enum FollowOption {
    local_only,
    if_no_local,
    always
};
struct LinkInfo {
    Lookup target;
    Register target_reg;
    FollowOption def_pass_on_follow_rule;
    FollowOption limiting_follow_rule;
};
```

The above information is set for each link when it is created. A link name is given to the link when it is created. The name uniquely identifies a link in a trader.

```
typedef Istring LinkName;
typedef sequence<LinkName> LinkNameSeq;
```

A link is unidirectional. Only the source trader is directly aware of a link; it is the source trader that supports the Link interface.

Additional information may be kept with a link to describe characteristics of the target trading service as perceived by the source trader.

Policies

Policies provide information to affect trader behavior at run time. Policies are represented as name value pairs.

```
typedef string PolicyName; // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
    PolicyName name;
    PolicyValue value;
};
```

```
typedef sequence<Policy> PolicySeq;
```

Some policies cannot be overridden, while other policies apply in the absence of further information and can be overridden. Policies can be grouped into two categories:

1. Policies that scope the extent of a search.
2. Policies that determine the functionality applied to an operation.

Different policies are associated with different roles in the performance of the trading function. These roles are:

T = Trader

L = Link

I = Import

Standardized Scoping Policies:

The following table lists the standardized scoping policies.

Table 16-2 Scoping Policies

Name	**Where**	**IDL Type**	**Description**
def_search_card	T	unsigned long	Default upper bound of offers to be searched; used if no search_card is specified.
max_search_card	T	unsigned long	Maximum upper bound of offers to be searched.
search_card	I	unsigned long	Nominated upper bound of offers to be searched; will be overridden by max_search_card.
def_match_card	T	unsigned long	Default upper bound of matched offers to be ordered; used if no match_card is specified.
max_match_card	T	unsigned long	Maximum upper bound of matched offers to be ordered.
match_card	I	unsigned long	Nominated upper bound of offers to be ordered; will be overridden by max_match_card.
def_return_card	T	unsigned long	Default upper bound of ordered offers to be returned; used if no return_card is specified.
max_return_card	T	unsigned long	Maximum upper bound of ordered offers to be returned.
return_card	I	unsigned long	Nominated upper bound of ordered offers to be returned; will be overridden by max_return_card.

Name	Where	IDL Type	Description
def_hop_count	T	unsigned long	Default upper bound of depth of links to be traversed if hop_count is not specified.
max_hop_count	T	unsigned long	Maximum upper bound of depth of links to be traversed.
hop_count	I	unsigned long	Nominated upper bound of depth of links to be traversed; will be overridden by the trader's max_hop_count.
def_pass_on_follow_rule	L	FollowOption	Default link-follow behavior to be passed on for a particular link if an importer does not specify its link_follow_rule; it must not exceed limiting_follow_rule.
limiting_follow_rule	L	FollowOption	Limiting link follow behavior for a particular link.
def_follow_policy	T	FollowOption	Default link follow behavior for a particular trader.
max_follow_policy	T	FollowOption	Limiting link follow policy for all links of the trader - overrides both link and importer policies.
max_link_follow_policy	T	FollowOption	Upper bound on the value of a link's limiting follow rule at the time of creation or modification of a link.
link_follow_rule	I	FollowOption	Nominated link follow behavior; it will be overridden by the trader's max_follow_policy and the link's limiting_follow_rule.
starting_trader	I	TraderName	An importer scopes its search by nominating that the query operation starts at a remote trader; a trader is obliged to forward the request down a link even if the link behavior is local_only.
request_id	I	OctetSeq	An identifier for a query operation initiated by a source trader acting as an importer on a link; a trader is not obliged to generate an id, but is obliged to pass one received down a link.
exact_type_match	I	boolean	If TRUE, only offers of exactly the service type specified by the importer are considered; if FALSE (or if unspecified), offers of any serviced type that conforms to the importer's service type are considered.

The IDL types for TraderName and OctetSeq are:

```
typedef LinkNameSeq TraderName;
typedef sequence<octet> OctetSeq;
```

The results received by an importer are affected by the scoping policies. The hop_count and link follow policies set the scope of the traders to visit. N1 is the total service offer space of those traders. Those offers that have conformant service type are gathered into the set N2; the actual size of N2 may be further restricted by the search cardinality policies. Constraints are applied to N2 to produce a set N3 of offers which satisfy both the service type and the constraints; N3 may be further restricted by the match cardinality policies. The set N3 is then ordered using preferences to produce the set N4. The final set of offers returned to the importer, N5, may be further reduced by the returned cardinality policies.

This is illustrated by the following diagram, where |N1| >= |N2| >= |N3| = |N4| >= |N5

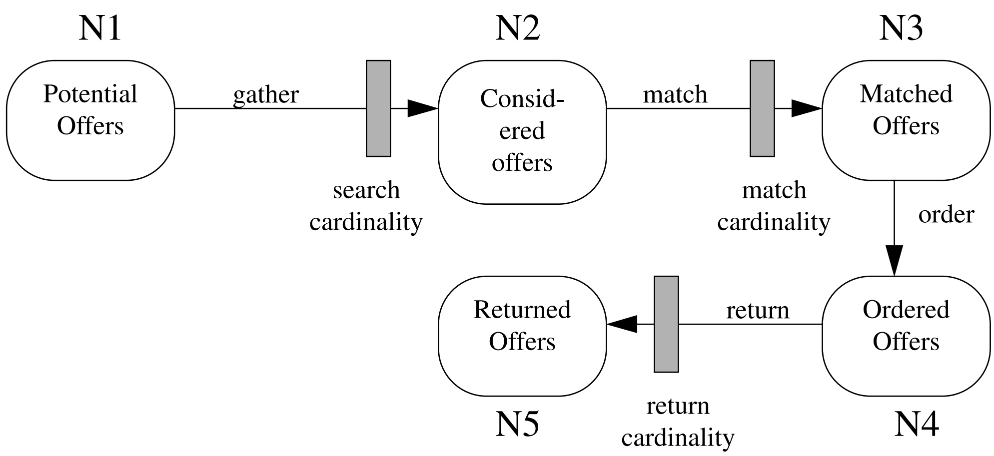


Figure 16-3 Pipeline View of Trader Query Steps and Cardinality Constraint Application

Standardized Capability Supported Policies

There are three optional capabilities (proxy offer, dynamic properties, and modify offers) that a trader may or may not wish to support. If a trader does not support a capability, then an importer cannot override it with its policy parameter. However, if a trader supports a capability and an importer does not wish to consider offers that require such functionality, then the trader must respect the importer's wish.

The following table lists the standardized policies related to supported functionality.

Table 16-3 Capability Supported Policies

Name	**Where**	**IDL Type**	**Description**
supports_modifiable_properties	T	boolean	Whether the trader supports property modification.
use_modifiable_properties	I	boolean	Whether to consider offers with modifiable properties in the search.
supports_dynamic_properties	T	boolean	Whether the trader supports dynamic properties.
use_dynamic_properties	I	boolean	Whether to consider offers with dynamic properties in the search.
supports_proxy_offers	T	boolean	Whether the trader supports proxy offers.
use_proxy_offers	I	boolean	Whether to consider proxy offers in the search.

Trader Policies

Policies can be set for a trader as a whole. Trader policies are defined as attributes of the trader object. They are specified initially when the trader is created, and can be modified/interrogated via the Admin interface. An importer can interrogate these trader policies via its Lookup interface. An exporter can interrogate a trader's functionality supported policies via its Register interface.

Link Follow Behavior

Each link in a trader has its own follow behavior policies. A trader has a limiting follow policy, max_follow_policy, that overrides all the links of that trader for any given query. Follow behavior policies are specified for each link when a link is created. These policies, def_pass_on_follow_rule and limiting_follow_rule, can be interrogated/modified via the Link interface. The values they can have are limited by another trader policy, max_link_follow_policy, at the time of creation or modification. An importer can specify a link_follow_rule in a query. In the absence of an importer's link_follow_rule, the trader's def_follow_policy is used.

After searching its local offers in response to a query, a trader must decide whether to propagate the query along its links and, if so, what value for the link_follow_rule to pass on in the policies argument.

Recall that the OMG IDL for FollowOption is:

```
enum FollowOption {
    local_only,
    if_no_local,
    always
};
```

where “local_only” indicates that the link is followed only by explicit navigation (“starting_trader” policy), “if_no_local” indicates that the link is followed only if there are no local offers that satisfy the query, and “always” has the obvious semantics. These values are ordered as follows:

local_only < if_no_local < always

The follow policy for a particular link is:

```
if the importer specified a link_follow_rule policy
    min(trader.max_follow_policy, link.limiting_follow_rule,
        query.link_follow_rule)
else
    min(trader.max_follow_policy, link.limiting_follow_rule,
        trader.def_follow_policy)
```

If this value is “if_no_local” and there were no local offers that match the query, the nested query is performed; if this value is “always,” the nested query is performed.

If the nested query is permitted by the above rule, then the following logic determines the value for the “link_follow_rule” policy to pass on to the linked trader.

```
If the importer specified a link_follow_rule policy
    pass on min(query.link_follow_rule, link.limiting_follow_rule,
                trader.max_follow_policy)
else
    pass on min(link.def_pass_on_follow_rule,
        trader.max_follow_policy)
```

Importer Policies

An importer can specify zero or more importer policies in its policy parameter. If an importer policy is not specified, then the trader uses its default policy. If an importer policy exceeds the limiting policy values set by the trader, then the trader overrides the importer expectations with its limiting policy value.

If a starting_trader policy parameter is used, trader implementations shall place this policy parameter as the first element of the sequence when forwarding the query request to linked traders.

Exporter Policies

There are no exporter policies specified in this standard.

Link Creation Policies

At the time that a link is created, the default and limiting follow rules associated with the link are specified. These rules can be constrained by the max_link_follow_policy of the trader.

The trader first checks to see that the default rule is less than or equal to the limiting rule. If not, then an exception is raised. It then compares the limiting rule against the trader's max_link_follow_policy, again raising an exception if the limiting rule is greater than the trader's max_link_follow_policy.

16.2.8 Interworking Mechanisms

Link Traversal Control

The flexible nature of trader linkage allows arbitrary directed graphs of traders to be produced. This can introduce two types of problem:

- A single trader can be visited more than once during a search due to it appearing on more than one path (i.e., distinct set of connected edges) leading from a trader.
- Loops can occur. The most trivial example of this is where two previously disjoint trader spaces decide to join by exchanging links. This can result in the first trader propagating a query to the second and then having it returned immediately via the reverse link.

To ensure that a search does not enter into an infinite loop, a hop_count is used to limit the depth of links to propagate a search. The hop_count is decremented by one before propagating a query to other traders. The search propagation terminates at the trader when the hop_count reaches zero.

To avoid the unproductive revisiting of a particular trader while performing a query, a RequestId can be generated by a source trader for each query operation that it initiates for propagation to a target trader. The trader attribute of request_id_stem is used to form RequestId.

```
typedef sequence<octet> OctetSeq;
attribute OctetSeq request_id_stem;
```

A trader remembers the RequestId of all recent interworking query operations that it has been asked to perform. When an interworking query operation is received, the trader checks this history and only processes the query if it is the operation's first appearance.

In order for this to work, the administrator for a set of federated traders must have initialized the respective request_id_stems to non-overlapping values.

The RequestId is passed in an importer's policy parameter on the query operation to the target trader. If the target trader does not support the use of the RequestId policy, the target trader need not process the RequestId, but must pass the RequestId onto the next linked trader if the search propagates further.

Federated Query Example

To propagate a query request in a trading graph, each source trader acts as a client to the Lookup interface of the target trader and passes its client's query operation to its target trader.

The following example illustrates the modification of hop count parameter as a query request passes through a set of linked traders in a trading graph. We assume that the link follow policies in the traders will result in "always" follow behavior.

1. A query request is invoked at the trading interface of T1 with an importer's hop count policy expressed as hop_count = 4. The trader scoping policy for T1 includes max_hop_count = 5. The resultant hop_count applied for the search (after the arbitration action that combines the trader policy and the importer policy) is hop_count = 4.
2. We assume that no match is found in T1 and the resulting follow policy is always. That is, T1 is to pass the request to T3. A modified importer hop_count policy of hop_count = 3 is used. The local trader scoping policy for T3 includes max_hop_count = 1 and the generation of T3_Request_id to avoid repeat or cyclic searches of the same traders. The resultant scoping policy applied for the search at T3 is hop_count = 1 and the T3_Request_id is stored.
3. Assuming that no match is found in T3 and the resulting follow policy is always, the modified scoping parameter for the query request at T4 is: hop_count = 0 and request_id = T3_Request_id.
4. Assuming that no match is found in T4. Even though the max_hop_count = 4 for T4, the search is not propagated further. An unsuccessful search result will be passed back to T3, to T1, and finally to the user at T1.

Of course, if a query request is completed successfully at any of the traders on the linked search path, then the list of matched service offers will be returned to the original user. Whether the query request is propagated through the remaining trading graph depends upon the link follow policies; in this case, where it is assumed to be always, the query will still visit all of the traders commensurate with the hop count policy.

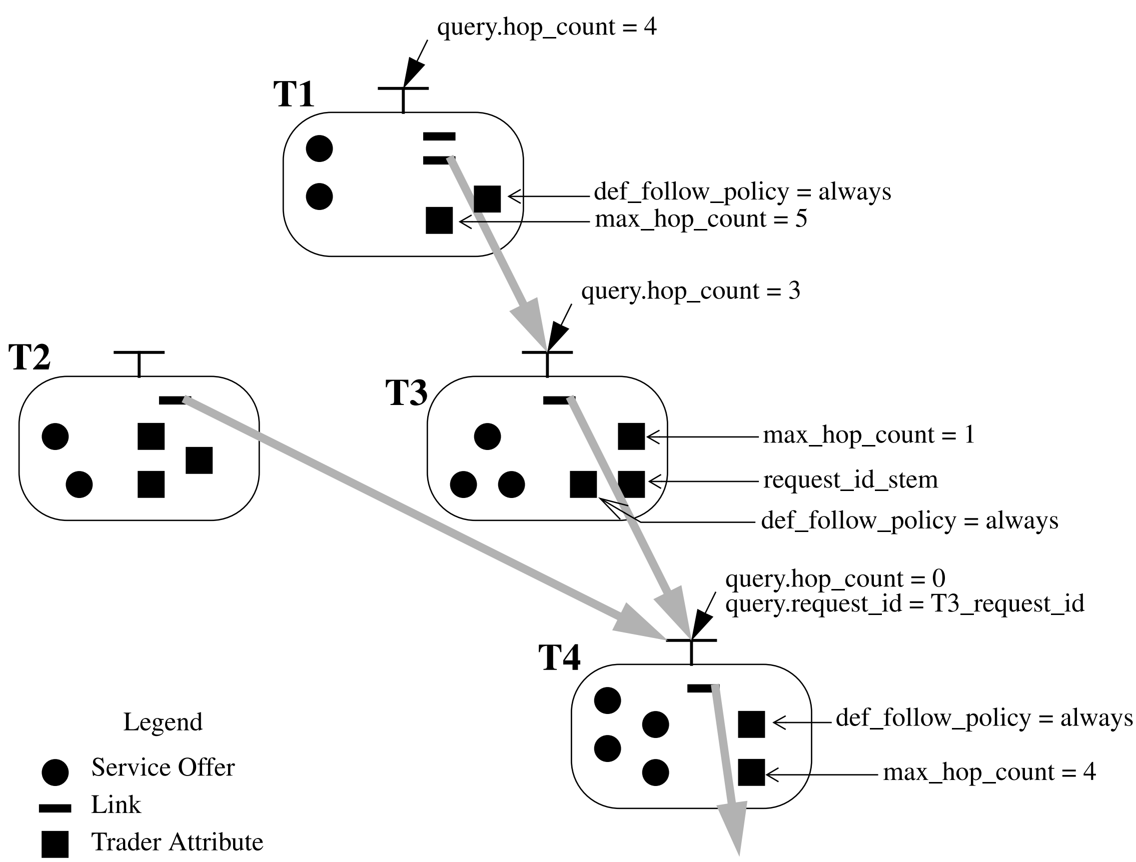


Figure 16-4 Flow of a query through a trader graph

Proxy Offers

A proxy offer is a cross between a service offer and a form of restricted link. It includes the service type and properties of a service offer and, as such, is matched in the same way. However, if the proxy offer matches the importer's requirements, rather than returning details of the offer, the query request (modified) is forwarded to the Lookup interface associated with the proxy offer.

```
typedef Istring ConstraintRecipe;

struct ProxyInfo {
    ServiceTypeName type;
    Lookup target;
    PropertySeq properties;
```

```
        boolean if_match_all;
        ConstraintRecipe recipe;
        PolicySeq policies_to_pass_on;
    };
```

If an importer's query results in a match to a proxy offer, the trader holding the proxy offer performs a nested query on the trader hiding behind the proxy offer with the following parameters:

- The original type parameter is passed on unchanged.
- A new constraint parameter is constructed following the ConstraintRecipe associated with the proxy offer.
- The original preference parameter is passed on unchanged.
- A new policies parameter is constructed by appending the policies_to_pass_on associated with the proxy offer to the original policies parameter.
- The original desired_props parameter is passed on unchanged.
- The calling trader supplies a value of how_many that makes sense given its resource constraints.

Proxy offers are a convenient way to package the encapsulation of a legacy system of "objects" into the trading system. It permits clients to lookup these "objects" by matching the proxy offer. The nested call to the proxy trader, together with the rewritten constraint expression and the additional policies appended to the original policy parameter, permits the dynamic creation of a service instance which encapsulates the legacy object. Another possible use of proxies is for a service factory to be advertised as a proxy offer; the nested call to the factory causes a new instance of the particular service to be manufactured.

A query may have matched a proxy offer due to a particular value of a property associated with the proxy offer. Any offer returned by the proxy trader as a result of the nested query must have the same value for that property so as not to violate the client's expectations regarding the constraint.

A trader does not have to support the proxy offer functionality. Traders that support such functionality must provide the Proxy interface for the export, withdraw, and describe of proxy offers. An importer can specify whether or not a trader should consider proxy offers during matching.

16.2.9 Trader Attributes

Each trader has its own characteristics, policies for supported functionalities, and policies for scoping the extent of search. These characteristics and policies are defined as attributes to the trader. These attributes are described in Table 16-4.

Table 16-4 Trader Attributes

Name	IDL Type	Description
def_search_card	unsigned long	Default upper bound of offers to be searched for a query operation
max_search_card	unsigned long	Maximum upper bound of offers to be searched for a query operation
def_match_card	unsigned long	Default upper bound of matched offers to be ordered in applying a preference criteria
max_match_card	unsigned long	Maximum upper bound of matched offers to be ordered in applying a preference criteria
def_return_card	unsigned long	Default upper bound of ordered offers to be returned to an importer
max_return_card	unsigned long	Maximum upper bound of ordered offers to be returned to an importer
def_hop_count	unsigned long	Default upper bound of depth of links to be traversed
max_hop_count	unsigned long	Maximum upper bound of depth of links to be traversed
max_list	unsigned long	The upper bound on the size of any list returned by the trader, namely the returned offers parameter in query, and the next_n operations in OfferIterator and OfferIdIterator.
def_follow_policy	FollowOption	Default link follow behavior for a particular trader
max_follow_policy	FollowOption	Limiting link follow policy for all links of the trader - overrides both link and importer policies
max_link_follow_policy	FollowOption	Most permissive follow policy allowed when creating new links
supports_modifiable_properties	boolean	Whether the trader supports property modification
supports_dynamic_properties	boolean	Whether the trader supports dynamic properties
supports_proxy_offers	boolean	Whether the trader supports proxy offers
type_repos	TypeRepository	Interface to trader's service type repository
request_id_stem	OctetSeq	Identification of the trader, to be used as the stem for the production of an id for a query request from one trader to another

These attributes are initially specified when a trader is created and can be modified/interrogated via the Admin interface.

16.3 Exceptions

This specification defines the exceptions raised by operations. Exceptions are parameterized to indicate the source of the error. The OMG IDL segments below refer to some of the typedef's defined in Section 16.2 Concepts and Data Types.

When multiple exception conditions arise, only one exception is raised. The choice of exception to raise is implementation-dependent.

16.3.1 For CosTrading module

Exceptions used in more than one interface

```
exception UnknownMaxLeft {};

exception NotImplemented {};

exception IllegalServiceType {
    ServiceTypeName type;
};

exception UnknownServiceType {
    ServiceTypeName type;
};

exception IllegalPropertyName {
    PropertyName name;
};

exception DuplicatePropertyName {
    PropertyName name;
};

exception PropertyTypeMismatch {
    ServiceTypeName type;
    Property prop;
};

exception MissingMandatoryProperty {
    ServiceTypeName type;
    PropertyName name;
```

```
};

exception IllegalConstraint {
    Constraint constr;
};

exception InvalidLookupRef {
    Lookup target;
};

exception IllegalOfferId {
    OfferId id;
};

exception UnknownOfferId {
    OfferId id;
};

exception ReadonlyDynamicProperty {
    ServiceTypeName type;
    PropertyName name;
};

exception DuplicatePolicyName {
    PolicyName name;
};
```

Additional Exceptions for Lookup Interface

```
exception IllegalPreference {
    Preference pref;
};

exception IllegalPolicyName {
    PolicyName name;
};

exception PolicyTypeMismatch {
    Policy the_policy;
```

```
};

exception InvalidPolicyValue {
    Policy the_policy;
};
exception IllegalPreference {
    Preference pref;
};
exception IllegalPolicyName {
    PolicyName name;
};
exception PolicyTypeMismatch {
    Policy policy;
};
```

Additional Exceptions For Register Interface

```
exception InvalidObjectRef {
    Object ref;
};

exception UnknownPropertyName {
    PropertyName name;
};

exception InterfaceTypeMismatch {
    ServiceTypeName type;
    Object reference;
};

exception ProxyOfferId {
    OfferId id;
};

exception MandatoryProperty {
    ServiceTypeName type;
    PropertyName name;
};

exception ReadonlyProperty {
    ServiceTypeName type;
```

```
    PropertyName name;
};

exception NoMatchingOffers {
    Constraint constr;
};

exception IllegalTraderName {
    TraderName name;
};

exception UnknownTraderName {
    TraderName name;
};

exception RegisterNotSupported {
    TraderName name;
};
```

Additional Exceptions for Link Interface

```
exception IllegalLinkName {
    LinkName name;
};

exception UnknownLinkName {
    LinkName name;
};

exception DuplicateLinkName {
    LinkName name;
};

exception DefaultFollowTooPermissive {
    FollowOption def_pass_on_follow_rule;
    FollowOption limiting_follow_rule;
};

exception LimitingFollowTooPermissive {
```

```
        FollowOption limiting_follow_rule;
        FollowOption max_link_follow_policy;
};
```

Additional Exceptions for Proxy Offer Interface

```
exception IllegalRecipe {
        ConstraintRecipe recipe;
};

exception NotProxyOfferId {
        OfferId id;
};
```

16.3.2 For CosTradingDynamic module

There is only a DynamicPropEval interface in this module. The interface has only one operation which raises the exception:

```
exception DPEvalFailure {
        CosTrading::PropertyName name;
        CORBA::TypeCode returned_type;
        any extra_info;
};
```

16.3.3 For CosTradingRepos module

There is only the ServiceTypeRepository interface in this module. The following interface-specific exceptions can be raised:

```
exception ServiceTypeExists {
        CosTrading::ServiceTypeName name;
};
exception InterfaceTypeMismatch {
        CosTrading::ServiceTypeName base_service;
        Identifier base_if;
        CosTrading::ServiceTypeName derived_service;
        Identifier derived_if;
};
exception HasSubTypes {
        CosTrading::ServiceTypeName the_type;
        CosTrading::ServiceTypeName sub_type;
};
exception AlreadyMasked {
```

```
        CosTrading::ServiceTypeName name;
};
exception NotMasked {
        CosTrading::ServiceTypeName name;
};
exception ValueTypeRedefinition {
        CosTrading::ServiceTypeName type_1;
        PropStruct definition_1;
        CosTrading::ServiceTypeName type_2;
        PropStruct definition_2;
};
exception DuplicateServiceTypeName {
        CosTrading::ServiceTypeName name;
};
```

16.4 Abstract Interfaces

To enable the construction of traders with varying support for the different trader interfaces, this specification defines several abstract interfaces from which each of the trading object service functional interfaces (Lookup, Register, Link, Proxy, and Admin) are derived. Each of these abstract interfaces are documented below.

16.4.1 TraderComponents

```
interface TraderComponents {

        readonly attribute Lookup lookup_if;
        readonly attribute Register register_if;
        readonly attribute Link link_if;
        readonly attribute Proxy proxy_if;
        readonly attribute Admin admin_if;
};
```

A trader's functionality can be configured by composing the defined interfaces in one of several prescribed combinations. The composition is not modeled through inheritance, but rather by multiple interfaces to an object. Given one of these interfaces, a way of finding the other associated interfaces is needed. To facilitate this, each trader functional interface is derived from the TraderComponents interface.

The TraderComponents interface contains five readonly attributes that provide a way to get a specific object reference.

The implementation of the _get_<interface>_if() operation must return a nil object reference if the trading service in question does not support that particular interface.

16.4.2 SupportAttributes

```
interface SupportAttributes {

    readonly attribute boolean supports_modifiable_properties;
    readonly attribute boolean supports_dynamic_properties;
    readonly attribute boolean supports_proxy_offers;
    readonly attribute TypeRepository type_repos;
};
```

In addition to the ability of a trader implementation to selectively choose which functional interfaces to support, a trader implementation may also choose not to support modifiable properties, dynamic properties, and/or proxy offers. The functionality supported by a trader implementation can be determined by querying the readonly attributes in this interface.

The type repository used by the trader implementation can also be obtained from this interface.

16.4.3 ImportAttributes

```
interface ImportAttributes {

    readonly attribute unsigned long def_search_card;
    readonly attribute unsigned long max_search_card;
    readonly attribute unsigned long def_match_card;
    readonly attribute unsigned long max_match_card;
    readonly attribute unsigned long def_return_card;
    readonly attribute unsigned long max_return_card;
    readonly attribute unsigned long max_list;
    readonly attribute unsigned long def_hop_count;
    readonly attribute unsigned long max_hop_count;
    readonly attribute FollowOption def_follow_policy;
    readonly attribute FollowOption max_follow_policy;
};
```

Each trader is configured with default and maximum values of certain cardinality and link follow constraints that apply to queries. The values for these constraints can be obtained by querying the attributes in this interface.

16.4.4 LinkAttributes

```
interface LinkAttributes {
    readonly attribute FollowOption max_link_follow_policy;
};
```

When a trader creates a new link or modifies an existing link the max_link_follow_policy attribute will determine the most permissive behavior that the link will be allowed. The value for this constraint on link creation and modification can be obtained from this interface.

16.5 Functional Interfaces

This section describes the five functional interfaces to a trading object service: Lookup, Register, Link, Admin, and Proxy. The two iterator interfaces needed for these functional interfaces are also described.

16.5.1 Lookup

```
interface Lookup:TraderComponents,SupportAttributes, ImportAttributes {

    typedef Istring Preference;

    enum HowManyProps {none, some, all };

    union SpecifiedProps switch (HowManyProps) {
        case some: PropertyNameSeq prop_names;
    };

    exception IllegalPreference {
        Preference pref;
    };

    exception IllegalPolicyName {
        PolicyName name;
    };

    exception PolicyTypeMismatch {
        Policy the_policy;
    };

    exception InvalidPolicyValue {
```

```
        Policy the_policy;
    };

    void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        IllegalPreference,
        IllegalPolicyName,
        PolicyTypeMismatch,
        InvalidPolicyValue,
        IllegalPropertyName,
        DuplicatePropertyName,
        DuplicatePolicyName
    );
};
```

Query Operation

Signature

```
    void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
```

```
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        IllegalPreference,
        IllegalPolicyName,
        PolicyTypeMismatch,
        InvalidPolicyValue,
        IllegalPropertyName,
        DuplicatePropertyName,
        DuplicatePolicyName
    );
```

Function

The query operation is the means by which an object can obtain references to other objects that provide services meeting its requirements.

The “type” parameter conveys the required service type. It is key to the central purpose of trading: to perform an introduction for future type safe interactions between importer and exporter. By stating a service type, the importer implies the desired interface type and a domain of discourse for talking about properties of the service.

- If the string representation of the “type” does not obey the rules for service type identifiers, then an IllegalServiceType exception is raised.
- If the “type” is correct syntactically but is not recognized as a service type within the trading scope, then an UnknownServiceType exception is raised.

The trader may return a service offer of a subtype of the “type” requested. Sub-typing of service types is discussed in “Service Types” on page 16-4. A service subtype can be described by the properties of its supertypes. This ensures that a well-formed query for the “type” is also a well-formed query with respect to any subtypes. However, if the importer specifies the policy of exact_type_match = TRUE, then only offers with the exact (no subtype) service type requested are returned.

The constraint “constr” is the means by which the importer states those requirements of a service that are not captured in the signature of the interface. These requirements deal with the computational behavior of the desired service, non-functional aspects, and non-computational aspects (such as the organization owning the objects that provide the service). An importer is always guaranteed that any returned offer satisfies the matching constraint at the time of import. If the “constr” does not obey the syntax rules for a legal constraint expression, then an IllegalConstraint exception is raised.

The “pref” parameter is also used to order those offers that match the “constr” so that the offers returned by the trader are in the order of greatest interest to the importer. If “pref” does not obey the syntax rules for a legal preference expression, then an IllegalPreference exception is raised.

The “policies” parameter allows the importer to specify how the search should be performed as opposed to what sort of services should be found in the course of the search. This can be viewed as parameterizing the algorithms within the trader implementation. The “policies” are a sequence of name-value pairs. The names available to an importer depend on the implementation of the trader. However, some names are standardized where they effect the interpretation of other parameters or where they may impact linking and federation of traders.

- If a policy name in this parameter does not obey the syntactic rules for legal PolicyName’s, then an IllegalPolicyName exception is raised.
- If the type of the value associated with a policy differs from that specified in this specification, then a PolicyTypeMismatch exception is raised.
- If subsequent processing of a PolicyValue yields any errors (e.g., the starting_trader policy value is malformed), then an InvalidPolicyValue exception is raised.
- If the same policy name is included two or more times in this parameter, then the DuplicatePolicyName exception is raised.

The “desired_props” parameter defines the set of properties describing returned offers that are to be returned with the object reference. There are three possibilities, the importer wants one of the properties, all of the properties (but without having to name them), or some properties (the names of which are provided).

- If any of the “desired_props” names do not obey the rules for identifiers, then an IllegalPropertyName exception is raised.
- If the same property name is included two or more times in this parameter, the DuplicatePropertyName exception is raised. The desired_props parameter may name properties which are not mandatory for the requested service type.
- If the named property is present in the matched service offer, then it shall be returned.

The desired_props parameter does not affect whether or not a service offer is returned. To avoid “missing” desired properties, the importer should specify “exists prop_name” in the constraint.

The returned offers are passed back in one of two ways (or a combination of both).

- The “offers” return result conveys a list of offers and the “offer_itr” is a reference to an interface at which offers can be obtained.
- The “how_many” parameter states how many offers are to be returned via the “offers” result, any remaining offers are available via the iterator interface. If the “how_many” exceeds the number of offers to be returned, then the “offer_itr” will be nil.

If any cardinality or other limits were applied by one or more traders in responding to a particular query, then the "limits_applied" parameter will contain the names of the policies which limited the query. The sequence of names returned in "limits_applied" from any federated or proxy queries must be concatenated onto the names of limits applied locally and returned.

Importer Policy Specifications

```
struct LookupPolicies {
    unsigned long search_card;
    unsigned long match_card;
    unsigned long return_card;
    boolean use_modifiable_properties;
    boolean use_dynamic_properties;
    boolean use_proxy_offers;
    TraderName starting_trader;
    FollowOption link_follow_rule;
    unsigned long hop_count;
    boolean exact_type_match;
};
```

The "search_card" policy indicates to the trader the maximum number of offers it should consider when looking for type conformance and constraint expression match. The lesser of this value and the trader's max_search_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_search_card attribute is used.

The "match_card" policy indicates to the trader the maximum number of matching offers to which the preference specification should be applied. The lesser of this value and the trader's max_match_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_match_card attribute is used.

The "return_card" policy indicates to the trader the maximum number of matching offers to return as a result of this query. The lesser of this value and the trader's max_return_card attribute is used by the trader. If this policy is not specified, then the value of the trader's def_return_card attribute is used.

The "use_modifiable_properties" policy indicates whether the trader should consider offers which have modifiable properties when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "use_dynamic_properties" policy indicates whether the trader should consider offers which have dynamic properties when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "use_proxy_offers" policy indicates whether the trader should consider proxy offers when constructing the set of offers to which type conformance and constraint processing should be applied. If the value of this policy is TRUE, then such offers will be included; if FALSE, they will not. If this policy is not specified, such offers will be included.

The "starting_trader" policy facilitates the distribution of the trading service itself. It allows an importer to scope a search by choosing to explicitly navigate the links of the trading graph. If the policy is used in a query invocation it is recommended that it be the first policy-value pair; this facilitates an optimal forwarding of the query operation. A "policies" parameter need not include a value for the "starting_trader" policy. Where this policy is present, the first name component is compared against the name held in each link. If no match is found, the InvalidPolicyValue exception is raised. Otherwise, the trader invokes query() on the Lookup interface held by the named link, but passing the "starting_trader" policy with the first component removed.

The "link_follow_rule" policy indicates how the client wishes links to be followed in the resolution of its query. See the discussion in "Link Follow Behavior" on page 16-16 for details.

The "hop_count" policy indicates to the trader the maximum number of hops across federation links that should be tolerated in the resolution of this query. The hop_count at the current trader is determined by taking the minimum of the trader's max_hop_count attribute and the importer's hop_count policy, if provided, or the trader's def_hop_count attribute if it is not. If the resulting value is zero, then no federated queries are permitted. If it is greater than zero, then it must be decremented before passing on to a federated trader.

The "exact_type_match" policy indicates to the trader whether the importer's service type must exactly match an offer's service type; if not (and by default), then any offer of a type conformant to the importer's service type is considered.

16.5.2 Offer Iterator

Signature

```
interface OfferIterator {
    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );
    boolean next_n (
```

```
        in unsigned long n,
        out OfferSeq offers
    );
    void destroy ();
};
```

Function

The OfferIterator interface is used to return a set of service offers from the query operation by enabling the service offers to be extracted by successive operations on the OfferIterator interface.

The next_n operation returns a set of service offers in the output parameter "offers." The operation returns n service offers if there are at least n service offers remaining in the iterator. If there are fewer than n service offers in the iterator, then all remaining service offers are returned. The actual number of service offers returned can be determined from the length of the "offers" sequence. The next_n operation returns TRUE if there are further service offers to be extracted from the iterator. It returns FALSE if there are no further service offers to be extracted.

The max_left operation returns the number of service offers remaining in the iterator. The exception UnknownMaxLeft is raised if the iterator cannot determine the remaining number of service offers (e.g., if the iterator determines its set of service offers through lazy evaluation).

The destroy operation destroys the iterator. No further operations can be invoked on an iterator after it has been destroyed.

16.5.3 Register

```
interface Register : TraderComponents, SupportAttributes {

    struct OfferInfo {
        Object reference;
        ServiceTypeName type;
        PropertySeq properties;
    };

    exception InvalidObjectRef {
        Object ref;
    };

    exception UnknownPropertyName {
        PropertyName name;
```

```
};

exception InterfaceTypeMismatch {
        ServiceTypeName type;
        Object reference;
};

exception ProxyOfferId {
        OfferId id;
};

exception MandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception ReadonlyProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception NoMatchingOffers {
        Constraint constr;
};

exception IllegalTraderName {
        TraderName name;
};

exception UnknownTraderName {
        TraderName name;
};

exception RegisterNotSupported {
        TraderName name;
};

OfferId export (
```

```
        in Object reference,
        in ServiceTypeName type,
        in PropertySeq properties
) raises (
        InvalidObjectRef,
        IllegalServiceType,
        UnknownServiceType,
        InterfaceTypeMismatch,
        IllegalPropertyName, // e.g. prop_name = "<foo-bar"
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        DuplicatePropertyName
);

void withdraw (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

OfferInfo describe (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

void modify (
        in OfferId id,
        in PropertyNameSeq del_list,
        in PropertySeq modify_list
) raises (
        NotImplemented,
        IllegalOfferId,
```

```
        UnknownOfferId,
        ProxyOfferId,
        IllegalPropertyName,
        UnknownPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MandatoryProperty,
        ReadonlyProperty,
        DuplicatePropertyName
    );

    void withdraw_using_constraint (
        in ServiceTypeName type,
        in Constraint constr
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        NoMatchingOffers
    );

    Register resolve (
        in TraderName name
    ) raises (
        IllegalTraderName,
        UnknownTraderName,
        RegisterNotSupported
    );
};
```

Export Operation

Signature

```
    OfferId export (
        in Object reference,
        in ServiceTypeName type,
        in PropertySeq properties
    ) raises (
```

```
        InvalidObjectRef,
        IllegalServiceType,
        UnknownServiceType,
        InterfaceTypeMismatch,
        IllegalPropertyName, // e.g. prop_name = "<foo-bar"
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        DuplicatePropertyName
    );
```

Function

The export operation is the means by which a service is advertised, via a trader, to a community of potential importers. The OfferId returned is the handle with which the exporter can identify the exported offer when attempting to access it via other operations. The OfferId is only meaningful in the context of the trader that generated it.

The "reference" parameter is the information that enables a client to interact with a remote server. If a trader implementation chooses to consider certain types of object references (e.g., a nil object reference) to be unexportable, then it may return the InvalidObjectRef exception in such cases.

The "type" parameter identifies the service type, which contains the interface type of the "reference" and a set of named property types that may be used in further describing this offer (i.e., it restricts what is acceptable in the properties parameter).

- If the string representation of the "type" does not obey the rules for identifiers, then an IllegalServiceType exception is raised.
- If the "type" is correct syntactically but a trader is able to unambiguously determine that it is not a recognized service type, then an UnknownServiceType exception is raised.
- If the trader can determine that the interface type of the "reference" parameter is not a subtype of the interface type specified in "type," then an InterfaceTypeMismatch exception is raised.

The "properties" parameter is a list of named values that conform to the property value types defined for those names. They describe the service being offered. This description typically covers behavioral, non-functional, and non-computational aspects of the service.

- If any of the property names do not obey the syntax rules for PropertyNames, then an IllegalPropertyName exception is raised.
- If the type of any of the property values is not the same as the declared type (declared in the service type), then a PropertyTypeMismatch exception is raised.

- If an attempt is made to assign a dynamic property value to a readonly property, then the ReadonlyDynamicProperty exception is raised.
- If the "properties" parameter omits any property declared in the service type with a mode of mandatory, then a MissingMandatoryProperty exception is raised.
- If two or more properties with the same property name are included in this parameter, the DuplicatePropertyName exception is raised.

Withdraw Operation

Signature

```
void withdraw (
    in OfferId id
) raises (
    IllegalOfferId,
    UnknownOfferId,
    ProxyOfferId
);
```

Function

The withdraw operation removes the service offer from the trader (i.e., after withdraw the offer can no longer be returned as the result of a query). The offer is identified by the "id" parameter which was originally returned by export.

- If the string representation of "id" does not obey the rules for offer identifiers, then an IllegalOfferId exception is raised.
- If the "id" is legal but there is no offer within the trader with that "id," then an UnknownOfferId exception is raised.
- If the "id" identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

Describe Operation

Signature

```
OfferInfo describe (
    in OfferId id
) raises (
    IllegalOfferId,
    UnknownOfferId,
    ProxyOfferId
);
```

Function

The describe operation returns the information about an offered service that is held by the trader. It comprises the "reference" of the offered service, the "type" of the service offer, and the "properties" that describe this offer of service. The offer is identified by the "id" parameter which was originally returned by export.

- If the string representation of "id" does not obey the rules for object identifiers, then an IllegalOfferId exception is raised.
- If the "id" is legal but there is no offer within the trader with that "id," then an UnknownOfferId exception is raised.
- If the "id" identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

Modify Operation

Signature

```
void modify (
    in OfferId id,
    in PropertyNameSeq del_list,
    in PropertySeq modify_list
) raises (
    NotImplemented,
    IllegalOfferId,
    UnknownOfferId,
    ProxyOfferId,
    IllegalPropertyName,
    UnknownPropertyName,
    PropertyTypeMismatch,
    ReadonlyDynamicProperty,
    MandatoryProperty,
    ReadonlyProperty,
    DuplicatePropertyName
);
```

Function

The modify operation is used to change the description of a service as held within a service offer. The object reference and the service type associated with the offer cannot be changed. This operation may:

- add new (non-mandatory) properties to describe an offer,
- change the values of some existing (not readonly) properties, or

- delete existing (neither mandatory nor readonly) properties.

The modify operation either succeeds completely or it fails completely. The offer is identified by the “id” parameter which was originally returned by export.

- If the string representation of “id” does not obey the rules for offer identifiers, then an IllegalOfferId exception is raised.
- If the “id” is legal but there is no offer within the trader with that “id,” then an UnknownOfferId exception is raised.
- If the “id” identifies a proxy offer rather than an ordinary offer, then a ProxyOfferId exception is raised.

The “del_list” parameter gives the names of the properties that are no longer to be recorded for the identified offer. Future query and describe operations will not see these properties.

- If any of the names within the “del_list” do not obey the rules for PropertyName’s, then an IllegalPropertyName exception is raised.
- If a “name” is legal but there is no property for the offer with that “name,” then an UnknownPropertyName exception is raised.
- If the list includes a property that has a mandatory mode, then the MandatoryProperty exception is raised.
- If the same property name is included two or more times in this parameter, the DuplicatePropertyName exception is raised.

The “modify_list” parameter gives the names and values of properties to be changed. If the property is not in the offer, then the modify operation adds it. The modified (or added) property values are returned in future query and describe operations in place of the original values.

- If any of the names within the “modify_list” do not obey the rules for PropertyName’s, then an IllegalPropertyName exception is raised.
- If the list includes a property that has a readonly mode, then the ReadonlyProperty exception is raised unless that readonly property is not currently recorded for the offer. The ReadonlyDynamicProperty exception is raised if an attempt is made to assign a dynamic property value to a readonly property.
- If the value of any modified property is of a type that is not the same as the type expected, then the PropertyTypeMismatch exception is raised.
- If two or more properties with the same property name are included in this argument, the DuplicatePropertyName exception is raised.

The NotImplemented exception shall be raised if and only if the supports_modifiable_properties attribute yields FALSE.

Note – It is not possible to change the service type of an offer or the object reference of the service. This has to be achieved by withdrawing and then re-exporting. The purpose of modify is to change the description of the offered service while preserving the OfferId. This might be important where the OfferId has been propagated around a community of objects.

Withdraw Using Constraint Operation

Signature

```
void withdraw_using_constraint (
        in ServiceTypeName type,
        in Constraint constr
) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        NoMatchingOffers
);
```

Function

The withdraw_using_constraint operation withdraws a set of offers from within a single trader. This set is identified in the same way that a query operation identifies a set of offers to be returned to an importer.

The “type” parameter conveys the required service type. Each offer of the specified type will have the constraint expression applied to it. If it matches the constraint expression, then the offer will be withdrawn.

- If “type” does not obey the rules for service types, then an IllegalServiceType exception is raised.
- If the “type” is correct syntactically but is not recognized as a service type by the trader, then an UnknownServiceType exception is raised.

The constraint “constr” is the means by which the client restricts the set of offers to those that are intended for withdrawal.

- If “constr” does not obey the syntax rules for a constraint then an IllegalConstraint exception is raised.
- If the constraint fails to match with any offer of the specified service type, then a NoMatchingOffers exception is raised.

Resolve Operation

Signature

```
Register resolve (
    in TraderName name
) raises (
    IllegalTraderName,
    UnknownTraderName,
    RegisterNotSupported
);
```

Function

This operation is used to resolve a context relative name for another trader. In particular, it is used when exporting to a trader that is known by a name rather than by an object reference. The client provides the name, which will be a sequence of name components.

- If the content of the parameter cannot yield legal syntax for the first component, then the IllegalTraderName exception is raised. Otherwise, the first name component is compared against the name held in each link.
- If no match is found, or the trader does not support links, the UnknownTraderName exception is raised. Otherwise, the trader obtains the register_if held as part of the matched link.
- If the Register interface is not nil, then the trader binds to the Register interface and invokes resolve but passes the TraderName with the first component removed; if it is nil, then the RegisterNotSupported exception is raised.

When a trader is able to match the first name component leaving no residual name, that trader returns the reference for the Register interface for that linked trader. In unwinding the recursion, intermediate traders return the Register interface reference to their client (another trader).

16.5.4 Offer Id Iterator

Signature

```
interface OfferIdIterator {

    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );
```

```
        boolean next_n (
            in unsigned long n,
            out OfferIdSeq ids
        );

        void destroy ();
    };
```

Function

The OfferIdIterator interface is used to return a set of offer identifiers from the list_offers operation and the list_proxies operation in the Admin interface by enabling the offer identifiers to be extracted by successive operations on the OfferIdIterator interface.

The next_n operation returns a set of offer identifiers in the output parameter "ids." The operation returns n offer identifiers if there are at least n offer identifiers remaining in the iterator. If there are fewer than n offer identifiers in the iterator, then all remaining offer identifiers are returned. The actual number of offer identifiers returned can be determined from the length of the "ids" sequence. The next_n operation returns TRUE if there are further offer identifiers to be extracted from the iterator. It returns FALSE if there are no further offer identifiers to be extracted.

The max_left operation returns the number of offer identifiers remaining in the iterator. The exception UnknownMaxLeft is raised if the iterator cannot determine the remaining number of offer identifiers (e.g., if the iterator determines its set of offer identifiers through lazy evaluation).

The destroy operation destroys the iterator. No further operations can be invoked on an iterator after it has been destroyed.

16.5.5 Admin

```
    interface Admin : TraderComponents, SupportAttributes,
                            ImportAttributes,LinkAttributes {

        typedef sequence<octet> OctetSeq;

        readonly attribute OctetSeq request_id_stem;

        unsigned long set_def_search_card (in unsigned long value);
        unsigned long set_max_search_card (in unsigned long value);

        unsigned long set_def_match_card (in unsigned long value);
        unsigned long set_max_match_card (in unsigned long value);
```

```
        unsigned long set_def_return_card (in unsigned long value);
        unsigned long set_max_return_card (in unsigned long value);

        unsigned long set_max_list (in unsigned long value);

        boolean set_supports_modifiable_properties (in boolean value);
        boolean set_supports_dynamic_properties (in boolean value);
        boolean set_supports_proxy_offers (in boolean value);

        unsigned long set_def_hop_count (in unsigned long value);
        unsigned long set_max_hop_count (in unsigned long value);

        FollowOption set_max_follow_policy (in FollowOption policy);
        FollowOption set_def_follow_policy (in FollowOption policy);

        FollowOption set_max_link_follow_policy (in FollowOption
                                                        policy);

        TypeRepository set_type_repos (in TypeRepository repository);

        OctetSeq set_request_id_stem (in OctetSeq stem);

        void list_offers (
            in unsigned long how_many,
            out OfferIdSeq ids,
            out OfferIdIterator id_itr
        ) raises (
            NotImplemented
        );

        void list_proxies (
            in unsigned long how_many,
            out OfferIdSeq ids,
            out OfferIdIterator id_itr
        ) raises (
            NotImplemented
        );
    };
```

Attributes and Set Operations

The admin interface enables the values of the trader attributes to be read and written. All attributes are defined as readonly in either SupportAttributes, ImportAttributes, LinkAttributes, or Admin. To set the trader “attribute” to a new value, set_<attribute_name> operations are defined in Admin. Each of these set operations returns the previous value of the attribute as its function value.

If the admin interface operation set_support_proxy_offers is invoked with a value set to FALSE in a trader which supports the proxy interface, the set_support_proxy_offer value does not affect the function of operations in the proxy interface. However, in this case, it does have the effect of making any proxy offers exported via the proxy interface for that trader unavailable to satisfy queries on that trader’s lookup interface.

List Offers Operation

Signature

```
void list_offers (
    in unsigned long how_many,
    out OfferIdSeq ids,
    out OfferIdIterator id_itr
) raises (
    NotImplemented
);
```

Function

The list_offers operation allows the administrator of a trader to perform housekeeping by obtaining a handle on each of the offers within a trader (e.g., for garbage collection etc.). Only the identifiers of ordinary offers are returned, identifiers of proxy offers are not returned via this operation. If the trader does not support the Register interface, the NotImplemented exception is raised.

The returned identifiers are passed back in one of two ways (or a combination of both).

- The “ids” return result conveys a list of offer identifiers and the “id_itr” is a reference to an interface at which additional offer identities can be obtained.
- The “how_many” parameter states how many identifiers are to be returned via the “ids” result; any remaining are available via the iterator interface. If the “how_many” exceeds the number of offers held in the trader, then the “id_itr” is nil.

List Proxies Operation

Signature

```
void list_proxies (
    in unsigned long how_many,
```

```
        out OfferIdSeq ids,
        out OfferIdIterator id_itr
    ) raises (
        NotImplemented
    );
```

Function

The list_proxies operation returns the set of offer identifiers for proxy offers held by a trader. Most "how_many" offer identifiers are returned via "ids" if:

- There are more than "how_many" offer identifiers, the remainder are returned via the "id_itr" iterator.
- There are only "how_many" or fewer offer identifiers, the id_itr is nil.
- The trader does not support the Proxy interface, the NotImplemented exception is raised.

16.5.6 Link

```
interface Link : TraderComponents, SupportAttributes,
                     LinkAttributes {

    struct LinkInfo {
        Lookup target;
        Register target_reg;
        FollowOption def_pass_on_follow_rule;
        FollowOption limiting_follow_rule;
    };

    exception IllegalLinkName {
        LinkName name;
    };

    exception UnknownLinkName {
        LinkName name;
    };

    exception DuplicateLinkName {
        LinkName name;
    };
```

```
exception DefaultFollowTooPermissive {
    FollowOption def_pass_on_follow_rule;
    FollowOption limiting_follow_rule;
};

exception LimitingFollowTooPermissive {
    FollowOption limiting_follow_rule;
    FollowOption max_link_follow_policy;
};

void add_link (
    in LinkName name,
    in Lookup target,
    in FollowOption def_pass_on_follow_rule,
    in FollowOption limiting_follow_rule
) raises (
    IllegalLinkName,
    DuplicateLinkName,
    InvalidLookupRef, // e.g. nil
    DefaultFollowTooPermissive,
    LimitingFollowTooPermissive
);

void remove_link (
    in LinkName name
) raises (
    IllegalLinkName,
    UnknownLinkName
);

LinkInfo describe_link (
    in LinkName name
) raises (
    IllegalLinkName,
    UnknownLinkName
);

LinkNameSeq list_links ();
```

```
        void modify_link (
            in LinkName name,
            in FollowOption def_pass_on_follow_rule,
            in FollowOption limiting_follow_rule
        ) raises (
            IllegalLinkName,
            UnknownLinkName,
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );
    };
```

Add_Link Operation

Signature

```
        void add_link (
            in LinkName name,
            in Lookup target,
            in FollowOption def_pass_on_follow_rule,
            in FollowOption limiting_follow_rule
        ) raises (
            IllegalLinkName,
            DuplicateLinkName,
            InvalidLookupRef, // e.g. nil
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );
```

Function

The add_link operation allows a trader subsequently to use the service of another trader in the performance of its own trading service operations.

The “name” parameter is used in subsequent link management operations to identify the intended link. If the parameter is not legally formed, then the IllegalLinkName exception is raised. An exception of DuplicateLinkName is raised if the link name already exists. The link name is also used as a component in a sequence of name components in naming a trader for resolving or forwarding operations. The sequence of context relative link names provides a path to a trader.

The "target" parameter identifies the Lookup interface at which the trading service provided by the target trader can be accessed. Should the Lookup interface parameter be nil, then an exception of InvalidLookupRef is raised. The target interface is used to obtain the associated Register interface, which will be subsequently returned as part of a describe_link operation and invoked as part of a resolve operation.

The "def_pass_on_follow_rule" parameter specifies the default link behavior for the link if no link behavior is specified on an importer's query request. If the "def_pass_on_follow_rule" exceeds the "limiting_follow_rule" specified in the next parameter, then a DefaultFollowTooPermissive exception is raised.

The "limiting_follow_rule" parameter specifies the most permissive link follow behavior that the link is willing to tolerate. The exception LimitingFollowTooPermissive is raised if this parameter exceeds the trader's attribute of "max_link_follow_policy" at the time of the link's creation. Note it is possible for a link's "limiting_follow_rule" to exceed the trader's "max_link_follow_policy" later in the life of a link, as it is possible that the trader could set its "max_link_follow_policy" to a more restrictive value after the creation of the link.

Remove Link Operation

Signature

```
void remove_link (
    in LinkName name
) raises (
    IllegalLinkName,
    UnknownLinkName
);
```

Function

The remove_link operation removes all knowledge of the target trader. The target trader cannot be used subsequently to resolve, forward, or propagate trading operations from this trader.

The "name" parameter identifies the link to be removed. The exception IllegalLinkName is raised if the link is formed poorly and the UnknownLinkName exception is raised if the named link is not in the trader.

Describe Link Operation

Signature

```
LinkInfo describe_link (
    in LinkName name
) raises (
    IllegalLinkName,
```

```
        UnknownLinkName
    );
```

Function

The describe_link operation returns information on a link held in the trader.

The “name” parameter identifies the link whose description is required. For a malformed link name, the exception IllegalLinkName is raised. An UnknownLinkName exception is raised if the named link is not found in the trader.

The operation returns a LinkInfo structure comprising:

- the Lookup interface of the target trading service,
- the Register interface of the target trading service, and
- the default, as well as the limiting follow behavior of the named link.

If the target service does not support the Register interface, then that field of the LinkInfo structure is nil. Given the description of the Register::resolve() operation in “Resolve Operation” on page 16-45, most implementations will opt for determining the Register interface when add_link is called and storing that information statically with the rest of the link state.

List Links Operation

Signature

```
    LinkNameSeq list_links ();
```

Function

The list_links operation returns a list of the names of all trading links within the trader. The names can be used subsequently for other management operations, such as describe_link or remove_link.

Modify Link Operation

Signature

```
    void modify_link (
        in LinkName name,
        in FollowOption def_pass_on_follow_rule,
        in FollowOption limiting_follow_rule
    ) raises (
        IllegalLinkName,
        UnknownLinkName,
```

```
        DefaultFollowTooPermissive,
        LimitingFollowTooPermissive
    );
```

Function

The modify_link operation is used to change the existing link follow behaviors of an identified link. The Lookup interface reference of the target trader and the name of the link cannot be changed.

The “name” parameter identifies the link whose follow behaviors are to be changed. A poorly formed “name” raises the IllegalLinkName exception. An UnknownLinkName exception is raised if the link name is not known to the trader.

The “def_pass_on_follow_rule” parameter specifies the new default link behavior for this link. If the “def_pass_on_follow_rule” exceeds the “limiting_follow_rule” specified in the next parameter, then a DefaultFollowTooPermissive exception is raised.

The “limiting_follow_rule” parameter specifies the new limit for the follow behavior of this link. The exception LimitingFollowTooPermissive is raised if the value exceeds the current “max_link_follow_policy” of the trader.

16.5.7 Proxy

```
interface Proxy: TraderComponents, SupportAttributes {
    typedef Istring ConstraintRecipe;
    struct ProxyInfo {
        ServiceTypeName type;
        Lookup target;
        PropertySeq properties;
        boolean if_match_all;
        ConstraintRecipe recipe;
        PolicySeq policies_to_pass_on;
    };
    exception IllegalRecipe {
        ConstraintRecipe recipe;
    };
    exception NotProxyOfferId {
        OfferId id;
    };
    OfferId export_proxy (
        in Lookup target,
        in ServiceTypeName type,
        in PropertySeq properties,
```

```
        in boolean if_match_all,
        in ConstraintRecipe recipe,
        in PolicySeq policies_to_pass_on
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        InvalidLookupRef, // e.g. nil
        IllegalPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        IllegalRecipe,
        DuplicatePropertyName,
        DuplicatePolicyName
    );

    void withdraw_proxy (
        in OfferId id
    ) raises (
        IllegalOfferId,
        UnknownOfferId,
        NotProxyOfferId
    );

    ProxyInfo describe_proxy (
        in OfferId id
    ) raises (
        IllegalOfferId,
        UnknownOfferId,
        NotProxyOfferId
    );
};
```

Export Proxy Operation

Signature

```
    OfferId export_proxy (
        in Lookup target,
```

```
        in ServiceTypeName type,
        in PropertySeq properties,

        in boolean if_match_all,
        in ConstraintRecipe recipe,
        in PolicySeq policies_to_pass_on
) raises (
        IllegalServiceType,
        UnknownServiceType,
        InvalidLookupRef, // e.g. nil
        IllegalPropertyName,
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        IllegalRecipe,
        DuplicatePropertyName,
        DuplicatePolicyName
);
```

Function

The Proxy interface enables the export and subsequent manipulation of proxy offers. Proxy offers enable run-time determination of the interface at which a service is provided. The export_proxy operation adds a proxy offer to the trader's set of service offers.

Like normal service offers, proxy offers have a service type "type" and named property values "properties." However, a proxy offer does not include an object reference at which the offered service is provided. Instead this object reference is obtained when it is needed for a query operation; it is obtained by invoking another query operation upon the "target" Lookup interface held in the proxy offer.

The "if_match_all" parameter, if TRUE, indicates that the trader should consider this proxy offer as a match to an importers query based upon type conformance alone (i.e., it does not match the importer's constraint expression against the properties associated with the proxy offer). This is most often useful when the constraint expression supplied by the importer is simply passed along in the secondary query operation.

The "recipe" parameter tells the trader how to construct the constraint expression for the secondary query operation to "target." The recipe language is described in Appendix C; it permits the secondary constraint expression to be made up of literals, values of properties of the proxy offer, and the primary constraint expression.

The “policies_to_pass_on” parameter provides a static set of <name, value> pairs for relaying on to the “target” trader. Table 16-5 describes how the secondary policy parameter is generated from the primary policy parameter and the “policies_to_pass_on.”

If a query operation matches the proxy offer (using the normal service type matching and property matching and preference algorithms), this primary query operation invokes a secondary query operation on the Lookup interface nominated in the proxy offer. Although the proxy offer nominates a Lookup interface, this interface is only required to conform syntactically to the Lookup interface; it need not conform to the Lookup interface behavior specified above.

The secondary query operation is detailed in Table 16-5.

Table 16-5 Primary/Secondary Policy Parameters

in ServiceTypeName type	The type is copied from primary query.
in Constraint constr	The recipe in the proxy offer is evaluated to provide the constr parameter.
in Preference pref	The preference is copied from the primary query.
in PolicySeq policies	The “policies” (names and values) contained in the policies_to_pass_on field of the proxy offer are appended to the policies of the primary query.
in SpecifiedProps desired_props	The desired_props are copied from the primary query.
in unsigned long how_many	The how_many parameter is set by the trader to reflect the trader implementation’s preference for receiving the resultant offer as a list or through an iterator.
out OfferSeq offers	At most how_many offers are returned from the secondary query operation via offers.
out OfferIterator offer_itr	If the secondary query needs to return more than how_many offers, then the remaining offers can be accessed via the iterator offer_itr. If there are only how_many or fewer offers, then offer_itr is nil.
out PolicyNameSeq limits_applied	The names of any policy limits that were applied by the proxy trader.

- The IllegalServiceType exception is raised if the service type name (type) is not well-formed.
- The UnknownServiceType exception is raised if the service type name (type) is not known to the trader.
- The InvalidLookupRef exception is raised if target is not a valid Lookup interface reference (e.g. if target is a nil object reference).
- The IllegalPropertyName exception is raised if a property name in “properties” is not well-formed.

- The PropertyTypeMismatch exception is raised if a property value is not of an appropriate type as determined by the service type.
- The ReadonlyDynamicProperty exception is raised if a dynamic property value was supplied for a property that was flagged as readonly.
- The MissingMandatoryProperty exception is raised if "properties" does not contain one of the mandatory properties defined by the service type.
- The IllegalRecipe exception is raised if the recipe is not well-formed.
- The DuplicatePropertyName exception is raised if two or more properties with the same property name are included in the "properties" parameter.
- The DuplicatePolicyName exception is raised if two or more policies with the same policy name are included in the "policies_to_pass_on" parameter.

Note – Proxy offers cannot be modified; they must be withdrawn and re-exported.

Withdraw Proxy Operation

Signature

```
void withdraw_proxy (
    in OfferId id
) raises (
    IllegalOfferId,
    UnknownOfferId,
    NotProxyOfferId
);
```

Function

The withdraw_proxy operation removes the proxy offer identified by "id" from the trader.

The IllegalOfferId exception is raised if "id" is not well-formed. The UnknownOfferId exception is raised if "id" does not identify any offer held by the trader. The NotProxyOfferId exception is raised if "id" identifies a normal service offer rather than a proxy offer.

Describe Proxy Operation

Signature

```
ProxyInfo describe_proxy (
    in OfferId id
) raises (
    IllegalOfferId,
```

```
        UnknownOfferId,
        NotProxyOfferId
    );
```

Function

The describe_proxy operation returns the information contained in the proxy offer identified by "id" in the trader.

The IllegalOfferId exception is raised if "id" is not well-formed. The UnknownOfferId exception is raised if "id" does not identify any offer held by the trader. The NotProxyOfferId exception is raised if "id" identifies a normal service offer rather than a proxy offer.

16.6 Service Type Repository

```
module CosTradingRepos {

    interface ServiceTypeRepository {

    // local types
        typedef sequence<CosTrading::ServiceTypeName>
                                ServiceTypeNameSeq;
        enum PropertyMode {
            PROP_NORMAL, PROP_READONLY,
            PROP_MANDATORY, PROP_MANDATORY_READONLY
        };
        struct PropStruct {
            CosTrading::PropertyName name;
            CORBA::TypeCode value_type;
            PropertyMode mode;
        };
        typedef sequence<PropStruct> PropStructSeq;

        typedef CosTrading::Istring Identifier; // IR::Identifier
        struct IncarnationNumber {
            unsigned long high;
            unsigned long low;
        };
        struct TypeStruct {
            Identifier if_name;
            PropStructSeq props;
```

```
        ServiceTypeNameSeq super_types;
        boolean masked;
        IncarnationNumber incarnation;
    };


    enum ListOption { all, since };
    union SpecifiedServiceTypes switch ( ListOption ) {
        case since: IncarnationNumber incarnation;
    };

// local exceptions
    exception ServiceTypeExists {
        CosTrading::ServiceTypeName name;
    };
    exception InterfaceTypeMismatch {
        CosTrading::ServiceTypeName base_service;
        Identifier base_if;
        CosTrading::ServiceTypeName derived_service;
        Identifier derived_if;
    };
    exception HasSubTypes {
        CosTrading::ServiceTypeName the_type;
        CosTrading::ServiceTypeName sub_type;
    };
    exception AlreadyMasked {
        CosTrading::ServiceTypeName name;
    };
    exception NotMasked {
        CosTrading::ServiceTypeName name;
    };
    exception ValueTypeRedefinition {
        CosTrading::ServiceTypeName type_1;
        PropStruct definition_1;
        CosTrading::ServiceTypeName type_2;
        PropStruct definition_2;
    };
    exception DuplicateServiceTypeName {
```

```
            CosTrading::ServiceTypeName name;
        };

    // attributes
        readonly attribute IncarnationNumber incarnation;

    // operation signatures
        IncarnationNumber add_type (
            in CosTrading::ServiceTypeName name,
            in Identifier if_name,
            in PropStructSeq props,
            in ServiceTypeNameSeq super_types
        ) raises (
            CosTrading::IllegalServiceType,
            ServiceTypeExists,
            InterfaceTypeMismatch,
            CosTrading::IllegalPropertyName,
            CosTrading::DuplicatePropertyName,
            ValueTypeRedefinition,
            CosTrading::UnknownServiceType,
            DuplicateServiceTypeName
        );

        void remove_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            HasSubTypes
        );

        ServiceTypeNameSeq list_types (
            in SpecifiedServiceTypes which_types
        );

        TypeStruct describe_type (
            in CosTrading::ServiceTypeName name
        ) raises (
```

```
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType
        );

        TypeStruct fully_describe_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType
        );

        void mask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            AlreadyMasked
        );

        void unmask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            NotMasked
        );

    };
}; /* end module CosTradingRepos */
```

Add Type Operation

Signature

```
IncarnationNumber add_type (
    in CosTrading::ServiceTypeName name,
    in Identifier if_name,
    in PropStructSeq props,
    in ServiceTypeNameSeq super_types
```

```
) raises (
        CosTrading::IllegalServiceType,
        ServiceTypeExists,
        InterfaceTypeMismatch,
        CosTrading::IllegalPropertyName,
        CosTrading::DuplicatePropertyName,
        ValueTypeRedefinition,
        CosTrading::UnknownServiceType,
        DuplicateServiceTypeName
);
```

Function

The add_type operation enables the creation of new service types in the service type repository. The caller supplies the "name" for the new type, the identifier for the interface associated with instances of this service type, the properties definitions for this service type, and the service type names of the immediate super-types to this service type.

If the type creation is successful, an incarnation number is returned as the value of the operation. Incarnation numbers are opaque values that are assigned to each modification to the repository's state. An incarnation number can be quoted when invoking the list_types operation to retrieve all changes to the service repository since a particular logical time. (Note: IncarnationNumber is currently declared as a struct consisting of two unsigned longs; what we really want here is an unsigned hyper [64-bit integer]. A future revision task force should modify this when CORBA systems support IDL 64-bit integers.)

- If the "name" parameter is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If the type already exists, then the ServiceTypeExists exception is raised.
- If the "if_name" parameter is not a sub-type of the interface associated with a service type from which this service type is derived, such that substitutability would be violated, then the InterfaceTypeMismatch exception is raised.
- If a property name supplied in the "props" parameter is malformed, the CosTrading::IllegalPropertyName exception is raised.
- If the same property name appears two or more times in the "props" parameter, the CosTrading::DuplicatePropertyName exception is raised.
- If a property value type associated with this service type illegally modifies the value type of a super-type's property, or if two super-types incompatibly declare value types for the same property name, then the ValueTypeRedefinition exception is raised.
- If one of the ServiceTypeNames in "super_types" is malformed, then the CosTrading::IllegalServiceType exception is raised.

- If one of the ServiceTypeNames in “super_types” does not exist, then the CosTrading::UnknownServiceType exception is raised.
- If the same service type name is included two or more times in this parameter, the DuplicateServiceTypeName exception is raised.

Remove Type Operation

Signature

```
void remove_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    HasSubTypes
);
```

Function

The remove_type operation removes the named type from the service type repository.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If “name” has a service type which has been derived from it, then the HasSubTypes exception is raised.

List Types Operation

Signature

```
ServiceTypeNameSeq list_types (
    in SpecifiedServiceTypes which_types
);
```

Function

The list_types operation permits a client to obtain the names of service types which are in the repository. The “which_types” parameter permits the client to specify one of two possible values:

- all types known to the repository
- all types added/modified since a particular incarnation number

The names of the requested types are returned by the operation for subsequent querying via the describe_type or the fully_describe_type operation.

Describe Type Operation

Signature

```
TypeStruct describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

Function

The describe_type operation permits a client to obtain the details for a particular service type.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.

Fully Describe Type Operation

Signature

```
TypeStruct fully_describe_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType
);
```

Function

The fully_describe_type operation permits a client to obtain the details for a particular service type. The property sequence returned in the TypeStruct includes all properties inherited from the transitive closure of its super types; the sequence of super types in the TypeStruct contains the names of the types in the transitive closure of the super type relation.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.

Mask Type Operation

Signature

```
void mask_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    AlreadyMasked
);
```

Function

The mask_type operation permits the deprecation of a particular type (i.e., after being masked, exporters will no longer be able to advertise offers of that particular type). The type continues to exist in the service repository due to other service types being derived from it.

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.
- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If the type is currently in the masked state, then the AlreadyMasked exception is raised.

Unmask Type Operation

Signature

```
void unmask_type (
    in CosTrading::ServiceTypeName name
) raises (
    CosTrading::IllegalServiceType,
    CosTrading::UnknownServiceType,
    NotMasked
);
```

Function

The unmask_type undeprecates a type (i.e., after being unmasked, exporters will be able to resume advertisement of offers of that particular type).

- If “name” is malformed, then the CosTrading::IllegalServiceType exception is raised.

- If “name” does not exist within the repository, then the CosTrading::UnknownServiceType exception is raised.
- If the type is not currently in the masked state, then the NotMasked exception is raised.

16.7 Dynamic Property Evaluation interface

```
module CosTradingDynamic {

    exception DPEvalFailure {
        CosTrading::PropertyName name;
        CORBA::TypeCode returned_type;
        any extra_info;
    };

    interface DynamicPropEval {

        any evalDP (
            in CosTrading::PropertyName name,
            in TypeCode returned_type,
            in any extra_info)
        raises (DPEvalFailure);
    };

    struct DynamicProp {
        DynamicPropEval eval_if;
        CORBA::TypeCode returned_type;
        any extra_info;
    };
};
```

The DynamicPropEval interface is provided by an exporter who wishes to provide a dynamic property value in a service offer held by the trader.

When exporting a service offer (or proxy offer), the property with the dynamic value has an “any” value which contains a DynamicProp structure rather than the normal property value. A trader which supports dynamic properties accepts this DynamicProp value as containing the information which enables a correctly-typed property value to be obtained during the evaluation of a query. The export (or export_proxy) operation raises the PropertyTypeMismatch if the returned_type is not appropriate for the property name as defined by the service type.

Readonly properties may not have dynamic values. The export and modify operations on the Register interface and the export_proxy operation on the Proxy interface raise the ReadonlyDynamicProperty exception if dynamic values are assigned to readonly properties.

When a query requires a dynamic property value, the evalDP operation is invoked on the eval_if interface in the DynamicProp structure. The property name parameter is the name of the property whose value is being obtained. The returned_type and extra_info parameters are copied from the DynamicProp structure. The evalDP operation returns an any value which should contain a value for that property. The value should be of a type indicated by returned_type.

The DPEvalFailure exception is raised if the value for the property cannot be determined. If the value is required for the evaluation of a constraint or preference, then that evaluation is deemed to have failed on that service offer (or proxy offer).

Other than the preceding rules, the behavior of the evalDP operation is not specified by this standard. In particular, the purpose of the extra_info data in determining the dynamic property value is implementation-specific.

If the trader does not support dynamic properties (indicated by the trader attribute supports_dynamic_properties), the export and export_proxy operations should not be parameterized by dynamic properties. The behavior of such traders in such circumstances is not specified by this standard.

If the trader does not support dynamic properties or the importer has requested that dynamic properties are not used (via the policies parameter of the query operation), then dynamic property evaluation is not performed. If the value of a dynamic property is required by the evaluation of a constraint or preference, then that evaluation is deemed to have failed on that service offer (or proxy offer).

The describe operation of the Register interface and the describe_proxy operation of the Proxy interface do not perform dynamic property evaluation, but return the DynamicProp structure as the value of the property. As these interfaces are used to create dynamic properties via the export and export_proxy operations, the other operations on these interfaces must ensure that the dynamic nature of the properties remains visible to the exporters.

The modify operation on the Register interface of a trader which supports dynamic properties must accept the establishment and modification of dynamic properties, consistent with the export operation. There is no restriction on a property value changing from a static value stored by the trader into a dynamic value, and vice versa.

Note – Readonly static properties may not be modified to be dynamic.

16.8 Conformance Criteria

The following interfaces are programmatic reference points for testing conformance:

- the Lookup interface (as server) provided by the trader implementation under test
- the Register interface (as server) provided by the trader implementation under test
- the Admin interface (as server) provided by the trader implementation under test
- the Link interface (as server) provided by the trader implementation under test
- the Proxy interface (as server) provided by the trader implementation under test
- a Lookup interface (as client) of a linked trader, used by the trader implementation under test

- a Register interface (as client) of a linked trader, used by the trader implementation under test
- a DynamicPropEval interface (as client) of an object, used by the trader implementation under test during the evaluation of a dynamic property

The behavior defined for each of the operations in the interface specifications shall be exhibited at the conformance points associated with that behavior.

The following taxonomy is defined for specific implementation conformance classes of trading object service implementations:

- query trader - supports the Lookup interface
- simple trader - supports the Lookup and Register interfaces
- stand-alone trader - supports the Lookup, Register, and Admin interfaces
- linked trader - supports the Lookup, Register, Admin, and Link interfaces; is also a client for Lookup and Register interfaces
- proxy trader - supports the Lookup, Register, Admin, and Proxy interfaces; is also a client for Lookup interfaces
- full-service trader - supports the Lookup, Register, Admin, Link, and Proxy interfaces; is also a client for Lookup and Register interfaces

Any of these specific trading object service classes may also be a client for the DynamicPropEval interface if it supports dynamic properties.

16.8.1 Conformance Requirements for Trading Interfaces as Server

Since the interfaces to a trading object service are separable, and support for those interfaces is selectable subject to the conformance classes defined above, this section specifies the conformance requirements on a per-interface basis.

Lookup Interface

An implementation claiming conformance to the Lookup interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Lookup interface as documented in "Lookup" on page 16-30.

An implementation claiming conformance to the Lookup interface as server shall also support the OfferIterator interface as server as documented in "Offer Iterator" on page 16-35.

Register Interface

An implementation claiming conformance to the Register interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Register interface as documented in "Register" on page 16-36, with the following permitted exceptions:

- An implementation which only allows the value of FALSE for the supports_modifiable_properties attribute is conformant, in which case it may reject a service offer which includes modifiable properties passed in an export operation, and may always respond to modify operation requests with an exception.
- An implementation which only allows the value of FALSE for the supports_dynamic_properties attribute is conformant, in which case it may reject a service offer which includes dynamic properties passed in an export operation.
- An implementation claiming conformance to the Register interface as server, with the value of the supports_dynamic_properties set to TRUE, shall be able to assume the client role for the DynamicPropEval interface.
- An implementation claiming conformance to the Register interface as server, with the value of the readonly attribute supports_proxy_offers set to TRUE, shall also support the Proxy interface.

Admin Interface

An implementation claiming conformance to the Admin interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Admin interface as documented in “Admin” on page 16-46.

An implementation claiming conformance to the Admin interface as server shall also support the OfferIdIterator interface as server as documented in “Offer Id Iterator” on page 16-45.

Link Interface

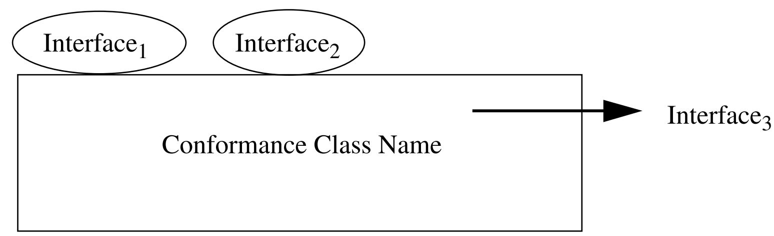
An implementation claiming conformance to the Link interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Link interface as documented in “Link” on page 16-49.

Proxy Interface

An implementation claiming conformance to the Proxy interface as server shall implement the complete behavior associated with all the operations and readonly attributes defined within the scope of the Proxy interface as documented in “Proxy” on page 16-54.

16.8.2 Conformance Requirements for Implementation Conformance Classes

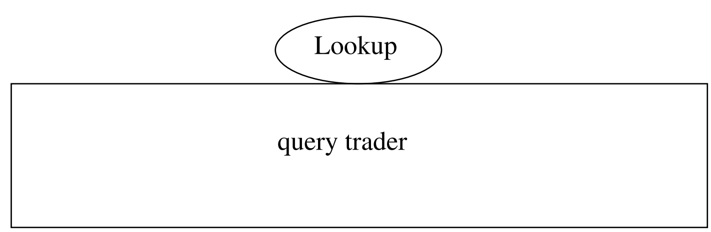
In the sections below, the following graphical notation is used:



The meaning of this notation is as follows:

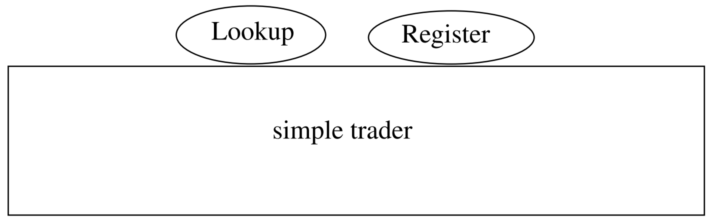
- The rectangle represents an implementation of "Conformance Class Name."
- The ellipses on the surface of the rectangle represent the interfaces supported by this implementation.
- The arrows to the right indicate that traders of this conformance class act as clients to other traders via the named interface.

Query Trader



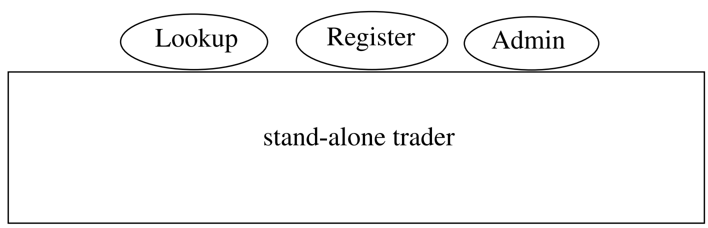
A trading object service implementation claiming conformance to the query trader conformance class shall meet the conformance requirements of the Lookup interface as server.

Simple Trader



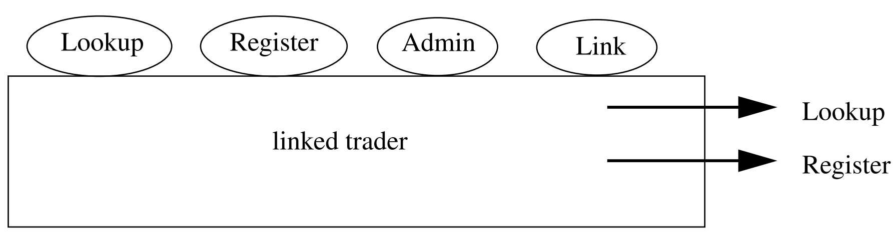
A trading object service implementation claiming conformance to the simple trader conformance class shall meet the conformance requirements of the Lookup and Register interfaces as server.

Stand-alone Trader



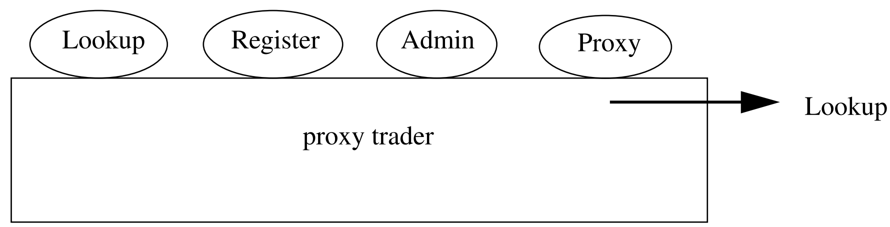
A trading object service implementation claiming conformance to the stand-alone trader conformance class shall meet the conformance requirements of the Lookup, Register, and Admin interfaces as server.

Linked Trader



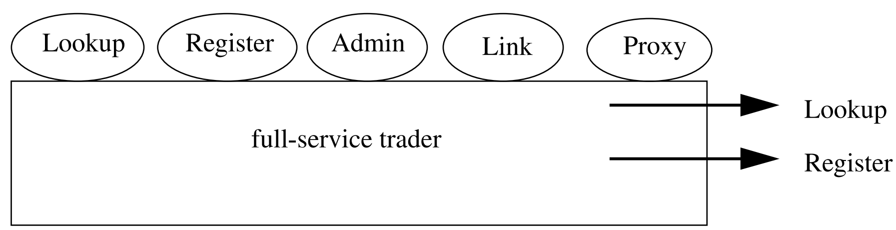
A trading object service implementation claiming conformance to the linked trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, and Link interfaces as server.

Proxy Trader



A trading object service implementation claiming conformance to the proxy trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, and Proxy interfaces as server.

Full-service Trader



A trading object service implementation claiming conformance to the full-service trader conformance class shall meet the conformance requirements of the Lookup, Register, Admin, Link, and Proxy interfaces as server.

Appendix A CORBA OMG IDL based Specification of the Trading Function

This appendix provides the CORBA OMG IDL specification of the interface signature for the trading function's computational specification. It specifies the signature for each computational operation in OMG IDL, according to the functional description (signature and semantics) provided in the body of this chapter.

A.1 OMG Trading Function Module

```
module CosTrading {

    // forward references to our interfaces

    interface Lookup;
    interface Register;
    interface Link;
    interface Proxy;
    interface Admin;
    interface OfferIterator;
    interface OfferIdIterator;

    // type definitions used in more than one interface

    typedef string Istring;
    typedef Object TypeRepository;

    typedef Istring PropertyName;
    typedef sequence<PropertyName> PropertyNameSeq;
    typedef any PropertyValue;
    struct Property {
        PropertyName name;
        PropertyValue value;
    };
    typedef sequence<Property> PropertySeq;

    struct Offer {
        Object reference;
        PropertySeq properties;
    };
    typedef sequence<Offer> OfferSeq;
```

```
typedef string OfferId;
typedef sequence<OfferId> OfferIdSeq;

typedef Istring ServiceTypeName;  // similar structure to IR::Identifier

typedef Istring Constraint;

enum FollowOption {
    local_only,
    if_no_local,
    always
};

typedef Istring LinkName;
typedef sequence<LinkName> LinkNameSeq;
typedef LinkNameSeq TraderName;

typedef string PolicyName;  // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
    PolicyName name;
    PolicyValue value;
};
typedef sequence<Policy> PolicySeq;

// exceptions used in more than one interface

exception UnknownMaxLeft {};

exception NotImplemented {};

exception IllegalServiceType {
    ServiceTypeName type;
};

exception UnknownServiceType {
```

```
        ServiceTypeName type;
};

exception IllegalPropertyName {
        PropertyName name;
};

exception DuplicatePropertyName {
        PropertyName name;
};
exception PropertyTypeMismatch {
        ServiceTypeName type;
        Property prop;
};

exception MissingMandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception ReadonlyDynamicProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception IllegalConstraint {
        Constraint constr;
};

exception InvalidLookupRef {
        Lookup target;
};

exception IllegalOfferId {
        OfferId id;
};

exception UnknownOfferId {
```

```
    OfferId id;
};

exception DuplicatePolicyName {
    PolicyName name;
};

// the interfaces

interface TraderComponents {

    readonly attribute Lookup lookup_if;
    readonly attribute Register register_if;
    readonly attribute Link link_if;
    readonly attribute Proxy proxy_if;
    readonly attribute Admin admin_if;
};

interface SupportAttributes {

    readonly attribute boolean supports_modifiable_properties;
    readonly attribute boolean supports_dynamic_properties;
    readonly attribute boolean supports_proxy_offers;
    readonly attribute TypeRepository type_repos;
};

interface ImportAttributes {

    readonly attribute unsigned long def_search_card;
    readonly attribute unsigned long max_search_card;
    readonly attribute unsigned long def_match_card;
    readonly attribute unsigned long max_match_card;
    readonly attribute unsigned long def_return_card;
    readonly attribute unsigned long max_return_card;
    readonly attribute unsigned long max_list;
    readonly attribute unsigned long def_hop_count;
    readonly attribute unsigned long max_hop_count;
    readonly attribute FollowOption def_follow_policy;
```

```
        readonly attribute FollowOption max_follow_policy;
};

interface LinkAttributes {

        readonly attribute FollowOption max_link_follow_policy;
};

interface Lookup:TraderComponents,SupportAttributes,ImportAttributes {

        typedef Istring Preference;

        enum HowManyProps { none, some, all };

        union SpecifiedProps switch ( HowManyProps ) {
                case some: PropertyNameSeq prop_names;
        };

        exception IllegalPreference {
                Preference pref;
        };

        exception IllegalPolicyName {
                PolicyName name;
        };

        exception PolicyTypeMismatch {
                Policy the_policy;
        };

        exception InvalidPolicyValue {
                Policy the_policy;
        };

        void query (
                in ServiceTypeName type,
                in Constraint constr,
                in Preference pref,
```

```
            in PolicySeq policies,
            in SpecifiedProps desired_props,
            in unsigned long how_many,
            out OfferSeq offers,
            out OfferIterator offer_itr,
            out PolicyNameSeq limits_applied
        ) raises (
            IllegalServiceType,
            UnknownServiceType,
            IllegalConstraint,
            IllegalPreference,
            IllegalPolicyName,
            PolicyTypeMismatch,
            InvalidPolicyValue,
            IllegalPropertyName,
            DuplicatePropertyName,
            DuplicatePolicyName
        );
    };

    interface Register : TraderComponents, SupportAttributes {

        struct OfferInfo {
            Object reference;
            ServiceTypeName type;
            PropertySeq properties;
        };

        exception InvalidObjectRef {
            Object ref;
        };

        exception UnknownPropertyName {
            PropertyName name;
        };

        exception InterfaceTypeMismatch {
            ServiceTypeName type;
```

```
        Object reference;
};

exception ProxyOfferId {
        OfferId id;
};

exception MandatoryProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception ReadonlyProperty {
        ServiceTypeName type;
        PropertyName name;
};

exception NoMatchingOffers {
        Constraint constr;
};

exception IllegalTraderName {
        TraderName name;
};

exception UnknownTraderName {
        TraderName name;
};

exception RegisterNotSupported {
        TraderName name;
};

OfferId export (
        in Object reference,
        in ServiceTypeName type,
        in PropertySeq properties
) raises (
```

```
        InvalidObjectRef,
        IllegalServiceType,
        UnknownServiceType,
        InterfaceTypeMismatch,
        IllegalPropertyName, // e.g. prop_name = "<foo-bar"
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MissingMandatoryProperty,
        DuplicatePropertyName
);

void withdraw (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

OfferInfo describe (
        in OfferId id
) raises (
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId
);

void modify (
        in OfferId id,
        in PropertyNameSeq del_list,
        in PropertySeq modify_list
) raises (
        NotImplemented,
        IllegalOfferId,
        UnknownOfferId,
        ProxyOfferId,
        IllegalPropertyName,
        UnknownPropertyName,
```

```
        PropertyTypeMismatch,
        ReadonlyDynamicProperty,
        MandatoryProperty,
        ReadonlyProperty,
        DuplicatePropertyName
    );

    void withdraw_using_constraint (
        in ServiceTypeName type,
        in Constraint constr
    ) raises (
        IllegalServiceType,
        UnknownServiceType,
        IllegalConstraint,
        NoMatchingOffers
    );

    Register resolve (
        in TraderName name
    ) raises (
        IllegalTraderName,
        UnknownTraderName,
        RegisterNotSupported
    );
};

interface Link : TraderComponents, SupportAttributes, LinkAttributes {

    struct LinkInfo {
        Lookup target;
        Register target_reg;
        FollowOption def_pass_on_follow_rule;
        FollowOption limiting_follow_rule;
    };

    exception IllegalLinkName {
        LinkName name;
    };
```

```
exception UnknownLinkName {
    LinkName name;
};

exception DuplicateLinkName {
    LinkName name;
};
exception DefaultFollowTooPermissive {
    FollowOption def_pass_on_follow_rule;
    FollowOption limiting_follow_rule;
};

exception LimitingFollowTooPermissive {
    FollowOption limiting_follow_rule;
    FollowOption max_link_follow_policy;
};

void add_link (
    in LinkName name,
    in Lookup target,
    in FollowOption def_pass_on_follow_rule,
    in FollowOption limiting_follow_rule
) raises (
    IllegalLinkName,
    DuplicateLinkName,
    InvalidLookupRef, // e.g. nil
    DefaultFollowTooPermissive,
    LimitingFollowTooPermissive
);

void remove_link (
    in LinkName name
) raises (
    IllegalLinkName,
    UnknownLinkName
);
```

```
        LinkInfo describe_link (
            in LinkName name
        ) raises (
            IllegalLinkName,
            UnknownLinkName
        );

        LinkNameSeq list_links ( );

        void modify_link (
            in LinkName name,
            in FollowOption def_pass_on_follow_rule,
            in FollowOption limiting_follow_rule
        ) raises (
            IllegalLinkName,
            UnknownLinkName,
            DefaultFollowTooPermissive,
            LimitingFollowTooPermissive
        );
    };

    interface Proxy : TraderComponents, SupportAttributes {

        typedef Istring ConstraintRecipe;

        struct ProxyInfo {
            ServiceTypeName type;
            Lookup target;
            PropertySeq properties;
            boolean if_match_all;
            ConstraintRecipe recipe;
            PolicySeq policies_to_pass_on;
        };

        exception IllegalRecipe {
            ConstraintRecipe recipe;
        };
```

```
exception NotProxyOfferId {
    OfferId id;
};

OfferId export_proxy (
    in Lookup target,
    in ServiceTypeName type,
    in PropertySeq properties,
    in boolean if_match_all,
    in ConstraintRecipe recipe,
    in PolicySeq policies_to_pass_on
) raises (
    IllegalServiceType,
    UnknownServiceType,
    InvalidLookupRef, // e.g. nil
    IllegalPropertyName,
    PropertyTypeMismatch,
    ReadonlyDynamicProperty,
    MissingMandatoryProperty,
    IllegalRecipe,
    DuplicatePropertyName,
    DuplicatePolicyName
);

void withdraw_proxy (
    in OfferId id
) raises (
    IllegalOfferId,
    UnknownOfferId,
    NotProxyOfferId
);

ProxyInfo describe_proxy (
    in OfferId id
) raises (
    IllegalOfferId,
    UnknownOfferId,
    NotProxyOfferId
```

```
        );
};

interface Admin : TraderComponents, SupportAttributes, ImportAttributes,
                                        LinkAttributes {

        typedef sequence<octet> OctetSeq;

        readonly attribute OctetSeq request_id_stem;

        unsigned long set_def_search_card (in unsigned long value);
        unsigned long set_max_search_card (in unsigned long value);

        unsigned long set_def_match_card (in unsigned long value);
        unsigned long set_max_match_card (in unsigned long value);

        unsigned long set_def_return_card (in unsigned long value);
        unsigned long set_max_return_card (in unsigned long value);

        unsigned long set_max_list (in unsigned long value);

        boolean set_supports_modifiable_properties (in boolean value);
        boolean set_supports_dynamic_properties (in boolean value);
        boolean set_supports_proxy_offers (in boolean value);

        unsigned long set_def_hop_count (in unsigned long value);
        unsigned long set_max_hop_count (in unsigned long value);

        FollowOption set_def_follow_policy (in FollowOption policy);
        FollowOption set_max_follow_policy (in FollowOption policy);

        FollowOption set_max_link_follow_policy (in FollowOption policy);

        TypeRepository set_type_repos (in TypeRepository repository);

        OctetSeq set_request_id_stem (in OctetSeq stem);

        void list_offers (
```

```
        in unsigned long how_many,
        out OfferIdSeq ids,
        out OfferIdIterator id_itr
    ) raises (
        NotImplemented
    );

    void list_proxies (
        in unsigned long how_many,
        out OfferIdSeq ids,
        out OfferIdIterator id_itr
    ) raises (
        NotImplemented
    );
};

interface OfferIterator {

    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );

    boolean next_n (
        in unsigned long n,
        out OfferSeq offers
    );

    void destroy ();
};

interface OfferIdIterator {
    unsigned long max_left (
    ) raises (
        UnknownMaxLeft
    );

    boolean next_n (
```

```
            in unsigned long n,
            out OfferIdSeq ids
        );

        void destroy ();
    };

}; /* end module CosTrading */
```

A.2 Dynamic Property Module

```
module CosTradingDynamic {

    exception DPEvalFailure {
        CosTrading::PropertyName name;
        CORBA::TypeCode returned_type;
        any extra_info;
    };

    interface DynamicPropEval {

        any evalDP (
            in CosTrading::PropertyName name,
            in CORBA::TypeCode returned_type,
            in any extra_info
        ) raises (
            DPEvalFailure
        );
    };

    struct DynamicProp {
        DynamicPropEval eval_if;
        CORBA::TypeCode returned_type;
        any extra_info;
    };
}; /* end module CosTradingDynamic */
```

A.3 Service Type Repository Module

```
module CosTradingRepos {
```

```
interface ServiceTypeRepository {

// local types
    typedef sequence<CosTrading::ServiceTypeName> ServiceTypeNameSeq;
    enum PropertyMode {
        PROP_NORMAL, PROP_READONLY,
        PROP_MANDATORY, PROP_MANDATORY_READONLY
    };
    struct PropStruct {
        CosTrading::PropertyName name;
        CORBA::TypeCode value_type;
        PropertyMode mode;
    };
    typedef sequence<PropStruct> PropStructSeq;

    typedef CosTrading::Istring Identifier;  // IR::Identifier
    struct IncarnationNumber {
        unsigned long high;
        unsigned long low;
    };
    struct TypeStruct {
        Identifier if_name;
        PropStructSeq props;
        ServiceTypeNameSeq super_types;
        boolean masked;
        IncarnationNumber incarnation;
    };

    enum ListOption { all, since };
    union SpecifiedServiceTypes switch ( ListOption ) {
        case since: IncarnationNumber incarnation;
    };

// local exceptions
    exception ServiceTypeExists {
        CosTrading::ServiceTypeName name;
    };
```

```
        exception InterfaceTypeMismatch {
            CosTrading::ServiceTypeName base_service;
            Identifier base_if;
            CosTrading::ServiceTypeName derived_service;
            Identifier derived_if;
        };
        exception HasSubTypes {
            CosTrading::ServiceTypeName the_type;
            CosTrading::ServiceTypeName sub_type;
        };
        exception AlreadyMasked {
            CosTrading::ServiceTypeName name;
        };
        exception NotMasked {
            CosTrading::ServiceTypeName name;
        };
        exception ValueTypeRedefinition {
            CosTrading::ServiceTypeName type_1;
            PropStruct definition_1;
            CosTrading::ServiceTypeName type_2;
            PropStruct definition_2;
        };
        exception DuplicateServiceTypeName {
            CosTrading::ServiceTypeName name;
        };

    // attributes
        readonly attribute IncarnationNumber incarnation;

    // operation signatures
        IncarnationNumber add_type (
            in CosTrading::ServiceTypeName name,
            in Identifier if_name,
            in PropStructSeq props,
            in ServiceTypeNameSeq super_types
        ) raises (
            CosTrading::IllegalServiceType,
            ServiceTypeExists,
```

```
        InterfaceTypeMismatch,
        CosTrading::IllegalPropertyName,
        CosTrading::DuplicatePropertyName,
        ValueTypeRedefinition,
        CosTrading::UnknownServiceType,
        DuplicateServiceTypeName
);

void remove_type (
        in CosTrading::ServiceTypeName name
) raises (
        CosTrading::IllegalServiceType,
        CosTrading::UnknownServiceType,
        HasSubTypes
);

ServiceTypeNameSeq list_types (
        in SpecifiedServiceTypes which_types
);

TypeStruct describe_type (
        in CosTrading::ServiceTypeName name
) raises (
        CosTrading::IllegalServiceType,
        CosTrading::UnknownServiceType
);

TypeStruct fully_describe_type (
        in CosTrading::ServiceTypeName name
) raises (
        CosTrading::IllegalServiceType,
        CosTrading::UnknownServiceType
);

void mask_type (
        in CosTrading::ServiceTypeName name
) raises (
        CosTrading::IllegalServiceType,
```

```
            CosTrading::UnknownServiceType,
            AlreadyMasked
        );

        void unmask_type (
            in CosTrading::ServiceTypeName name
        ) raises (
            CosTrading::IllegalServiceType,
            CosTrading::UnknownServiceType,
            NotMasked
        );

    };
}; /* end module CosTradingRepos */
```

Appendix B OMG Constraint Language BNF

This appendix provides the BNF specification of the CORBA standard constraint language; it is used for specifying both the constraint and preference expression parameters to various operations in the trader interfaces.

A statement in this language is an Istring. Other constraint languages may be supported by a particular trader implementation; the constraint language used by a client of the trader is indicated by embedding “<<Identifier major.minor>>” at the beginning of the string. If such an escape is not used, it is equivalent to embedding “<<OMG 1.0>>” at the beginning of the string.

B.1 Language Basics

B.1.1 Basic Elements

Both the constraint and preference expressions in a query can be constructed from property names of conformant offers and literals. The constraint language in which these expressions are written consists of the following items (examples of these expressions are shown in square brackets below each bulleted item):

- comparative functions: == (equality), != (inequality), >, >=, <, <=, ~ (substring match), in (element in sequence); the result of applying a comparative function is a boolean value
  [“Cost < 5” implies only consider offers with a Cost property value less than 5; “’Visa’ in CreditCards” implies only consider offers in which the CreditCards property, consisting of a set of strings, contains the string ’Visa’]
- boolean connectives: and, or, not
  [“Cost >= 2 and Cost <= 5” implies only consider offers where the value of the Cost property is in the range 2 <= Cost <= 5]
- property existence: exist
- property names
- numeric and string constants
- mathematical operators: +, -, *, /
  [“10 < 12.3 * MemSize + 4.6 * FileSize” implies only consider offers for which the arithmetic function in terms of the value of the MemSize and FileSize properties exceeds 10]
- grouping operators: (,)

Note that the keywords in the language are case sensitive.

B.1.2 Precedence Relations

The following precedence relations hold in the absence of parentheses, in the order of highest to lowest:

() exist unary-minus

not

* /

+ -

~

in

== != < <= > >=

and

or

B.1.3 Legal Property Value Types

While one can define properties of service types with arbitrarily complex OMG IDL value types, only the following property value types can be manipulated using the constraint language:

- boolean, short, unsigned short, long, unsigned long, float, double, char, Ichar, string, Istring
- sequences of the above types

The “exist” operator can be applied to any property name, regardless of the property’s value type.

B.1.4 Operator Restrictions

exist	can be applied to any property
~	can only be applied if left operand and right operand are both strings or both Istrings
in	can only be applied if the left operand is one of the simple types described above and the right operand is a sequence of the same simple type
==	can only be applied if the left and right operands are of the same simple type
!=	can only be applied if the left and right operands are of the same simple type
<	can only be applied if the left and right operands are of the same simple type
<=	can only be applied if the left and right operands are of the same simple type
>	can only be applied if the left and right operands are of the same simple type
>=	can only be applied if the left and right operands are of the same simple type
+	can only be applied to simple numeric operands
-	can only be applied to simple numeric operands
*	can only be applied to simple numeric operands

/	can only be applied to simple numeric operands

<, <=, >, >= comparisons imply use of the appropriate collating sequence for characters and strings; TRUE is greater than FALSE for booleans.

B.1.5 Representation of Literals

boolean	TRUE or FALSE
integers	sequences of digits, with a possible leading + or -
floats	digits with decimal point, with optional exponential notation
characters	char and Ichar are of the form '<char>', string and Istring are of the form '<char><char>+'; to embed an apostrophe in a string, place a backslash (\) in front of it; to embed a backslash in a string, use \\.

B.2 The Constraint Language BNF

B.2.1 The Constraint Language Proper in Terms of Lexical Tokens

```
<constraint>:=/* empty */
        |       <bool>

<preference>:=/* <empty> */
        |       min <bool>
        |       max <bool>
        |       with <bool>
        |       random
        |       first

<bool>:=   <bool_or>

<bool_or>:=<bool_or> or <bool_and>
        |       <bool_and>

<bool_and>:=<bool_and> and <bool_compare>
        |       <bool_compare>

<bool_compare>:=<expr_in> == <expr_in>
        |       <expr_in> != <expr_in>
        |       <expr_in> <  <expr_in>
        |       <expr_in> <= <expr_in>
```

| <expr_in> > <expr_in>
| <expr_in> >= <expr_in>
| <expr_in>

<expr_in>:=<expr_twiddle> in <Ident>
| <expr_twiddle>

<expr_twiddle>:=<expr> ~ <expr>
| <expr>

<expr>:= <expr> + <term>
| <expr> - <term>
| <term>

<term>:= <term> * <factor_not>
| <term> / <factor_not>
| <factor_not>

<factor_not>:=not <factor>
| <factor>

<factor>:=(<bool_or>)
| exist <Ident>
| <Ident>
| <Number>
| - <Number>
| <String>
| TRUE
| FALSE

B.2.2 "BNF" for Lexical Tokens up to Character Set Issues

<Ident>:= <Leader> <FollowSeq>

<FollowSeq>:=/* <empty> */
| <FollowSeq> <Follow>

<Number>:=<Mantissa>
| <Mantissa> <Exponent>

```
<Mantissa>:=<Digits>
        |       <Digits> .
        |       . <Digits>
        |       <Digits> . <Digits>

<Exponent>:=<Exp> <Sign> <Digits>

<Sign>:=  +
        |       -

<Exp>:=   E
        |       e

<Digits>:=<Digits> <Digit>
        |       <Digit>

<String>:=' <TextChars> '

<TextChars>:=/* <empty> */
        |       <TextChars> <TextChar>

<TextChar>:=<Alpha>
        |       <Digit>
        |       <Other>
        |       <Special>

<Special>:=\\
        |       \'
```

B.2.3 Character Set Issues

The previous BNF has been complete up to the non-terminals <Leader>, <Follow>, <Alpha>, <Digit>, and <Other>. For a particular character set, one must define the characters which make up these character classes.

Each character set which the trading service is to support must define these character classes. This appendix defines these character classes for the ASCII character set.

```
<Leader>:=<Alpha>
```

<Follow>:=<Alpha>

| <Digit>

| _

<Alpha> is the set of alphabetic characters [A-Za-z]

<Digit> is the set of digits [0-9]

<Other> is the set of ASCII characters that are not <Alpha>, <Digit>, or <Special>

Appendix C OMG Constraint Recipe Language

This appendix describes the recipe language used to construct the secondary constraint expression when resolving proxy offers; the secondary constraint expression is constructed from the primary constraint expression and the properties associated with the proxy offer.

A statement in this language is an Istring. Other recipe languages may be supported by a particular trader implementation; the recipe language used by a client of the trader is indicated by embedding "<<Identifier major.minor>>" at the beginning of the string. If such an escape is not used, it is equivalent to embedding "<<OMG 1.0>>" at the beginning of the string.

While the nested invocation of the Trader behind the proxy assumes support for the Lookup interface, the secondary constraint expression does not necessarily need to conform to the language described in Appendix B.

C.1 The Recipe Syntax

The rewriting from primary to secondary works similarly to formatted output in a variety of programming languages and systems. It is patterned after the variable replacement syntax of the Bourne and Korn shells on most UNIX systems.

When it is time to construct the secondary constraint expression from the recipe, the algorithm is as follows:

```
while not end of recipe
    fetch the next character from the recipe
    if not a '$' character
        append the character to the secondary constraint
    else
        fetch next character from the recipe
        if a '*' character
            append the entire primary constraint to the secondary constraint
        else if not a '(' character
            append the character to the secondary constraint
        else
            collect characters up to a ')' character, discarding ')'
            lookup property with that name
            append formatted value of that property to secondary constraint
```

C.2 Example

Assume a proxy offer has been exported to a trader with the following properties:

<Name, 'MyName'>, <Cost, 42>, <Host, 'x.y.co.uk'>

and with the following recipe:

"Name == $(Name) and Cost == $$$(Cost)"

The above algorithm will generate the following secondary constraint for the nested call to the trader behind the proxy:

"Name == 'MyName' and Cost == $42"

Object Collection Specification 17

The adopted specification used to create this chapter was OMG document ORBOS 96-07-09, July 1996. This chapter provides complete documentation for the Object Collection Service specification.

Contents

This chapter contains the following sections.

Section Title	Page
"Overview"	17-2
"Service Structure"	17-2
"Combined Collections"	17-10
"Restricted Access Collections"	17-14
"The CosCollection Module"	17-15
Appendix A, "OMG Object Query Service"	17-122
Appendix B, "Relationship to Other Relevant Standards"	17-131
Appendix C, "References"	17-136

17.1 Overview

Collections support the grouping of objects and support operations for the manipulation of the objects as a group. Common collection types are queues, sets, bags, maps, etc. Collection types differ in the "nature of grouping" exposed to the user. "Nature of grouping" is reflected in the operations supported for the manipulation of objects as members of a group. Collections, for example, can be ordered and thus support access to an element at position "i" while other collections may support associative access to elements via a key. Collections may guarantee the uniqueness of elements while others allow multiple occurrences of elements. A user chooses a collection type that matches the application requirements based on manipulation capabilities.

Collections are foundation classes used in a broad range of applications; therefore, they have to meet the general requirement to be able to collect elements of arbitrary type. On the other hand, a collection instance usually is a homogenous collection in the sense that all elements collected are of the same type, or support the same single interface.

Sometimes you may not want to do something to all elements in a collection, but only treat an individual object or traverse a collection explicitly (not implicitly via a collection operation). To enable this, a pointer abstraction often called an iterator is supported with collections. For example, an iterator points to an element in a collection and processes the element pointed to. Iterators can be moved and used to visit elements of a collection in an application defined manner. There can be many iterators pointing to elements of the same collection instance.

Normally, when operating on all elements of a collection, you want to pass user-defined information to the collection implementation about what to do with the individual elements or which elements are to be processed. To enable this, function interfaces are used. A collection implementation can rely on and use the defined function interface. A user has to specialize and implement these interfaces to pass the user-defined information to the implementation. A function interface can be used to pass element type specific information such as how to compare elements or pass a "program" to be applied to all elements.

17.2 Service Structure

The purpose of an Object Collection Service is to provide a uniform way to create and manipulate the most common collections generically. The Object Service defines three categories of interfaces to serve this purpose.

1. **Collection interfaces** and **collection factories**. A client chooses a collection interface which offers grouping properties that match the client's needs. A client creates a collection instance of the chosen interface using a collection factory. When creating a collection, a client has to pass element type specific information such as how to compare elements, how to test element equality, or the type checking desired. A client uses collections to manipulate elements as a group. When a

collection is no longer used it may be destroyed - this includes removing the elements collected, destroying element type specific information passed, and the iterators pointing to this collection.

2. **Iterator interfaces**. A client creates an iterator using the collection for which it is created as factory. A client uses an iterator to traverse the collection in an application defined manner, process elements pointed to, mark ranges, etc. When a client no longer uses an iterator, it destroys the iterator.

3. **Function interfaces**. A client creates user-defined specializations of these interfaces using user-defined factories. Instances are passed to a collection implementation when the collection is created (element type specific information) or as a parameter of an operation (for example, code to be executed for each element of the collection). Instances of function interfaces are used by a collection implementation rather than by a client.

17.2.1 Combined Property Collections

The Object Collection Service (or simply Collection Service) defined in this specification aims at being a complete and differentiated offering of interfaces supporting the grouping of objects. It enables a user to make a choice when following the rule "pay only for what you use." With this goal in mind, a very systematic approach was chosen.

Groups, or collections of objects, support operations and exhibit specific behaviors that are mainly related to the nature of the collection rather than the type of objects they collect.

"Nature of the collection" can be expressed in terms of well defined properties.

Ordering of elements

A *previous* or *nex*t relationship exists between the elements of an *ordered collection* which is exposed in the interface.

Ordering can be sequential or sorted. A sequential ordering can be explicitly manipulated; however, a sorted ordering is to be maintained implicitly based on a sort criteria to be defined and passed to the implementation by the user.

Access by key

A *key collection* allows associative access to elements via a *key*. A key can be computed from an element value via a user-defined key operation. Furthermore, key collections require key equality to be defined.

Element equality

An *equality collection* exploits the property that a test for element equality is defined (i.e., it can be tested whether an element is equal to another in terms of a user-defined element equality operation). This enables a test on containment, for example.

Uniqueness of entries

A collection with *unique* entries allows exactly one occurrence of an element key value, not *multiple* occurrences.

Meaningful combinations of these basic properties define "collections of differing nature of grouping." Table 17-1 provides an overview of meaningful combinations. The listed combinations are described in more detail in the following section.

Table 17-1 Interfaces derived from combinations of collection properties

		Unordered		Ordered		
				Sorted		Sequential
		Unique	Multiple	Unique	Multiple	Multiple
Key (Key equality must be specified)	**Element Equality**	**Map**	**Relation**	**Sorted Map**	**Sorted Relation**	
	No Element Equality	**KeySet**	**KeyBag**	**Key Sorted Set**	**Key SortedBag**	
No Key	**Element Equality**	**Set**	**Bag**	**SortedSet**	**Sorted Bag**	**Equality Sequence**
	No Element Equality		**Heap**			**Sequence**

Properties are mapped to interfaces - each interface assembling operations that exploit these properties. These interfaces are combined via multiple inheritance and form an *abstract interface hierarchy.* Abstract means that no instance of such a class can be instantiated, an attempt to do so may raise an exception at run-time. Leaves of this hierarchy represent concrete interfaces listed in the table above and can be instantiated by a user. They form a complete and differentiated offering of collection interfaces.

Restricted Access Collections

Common data structures based on these properties sometimes restrict access such as queues, stacks, or priority queues. They can be considered as restricted access variants of Sequence or KeySortedBag. These interfaces form their own hierarchy of *restricted access interfaces*. They are not incorporated into the hierarchy of combined properties because a user of restricted access interfaces should not be bothered with inherited operations which cannot be used in these interfaces. Nevertheless, to support several "views" on an interface, a restricted users view of a queue and an unrestricted system administrators view to the same queue instance, the restricted access collections are defined in a way that allows combining them with the combined properties collections via multiple inheritance.

All collections are unbounded (there is no explicit bound set) and controlled by the collections; however, it depends on the quality of service delivered whether there are “natural” limits such as the size of the paging space.

Collection Factories

For each concrete collection interface specified in this specification there is one corresponding collection factory defined. Each such factory offers a typed create operation for the creation of collection instances supporting the respective collection interface.

Additionally, a generic extensible factory is specified to enable the usage of many implementation variants for the same collection interface. This extensible generic factory allows the registration of implementation variants and their user-controlled selection at collection creation time.

Information to be passed to a collection at creation time is the element and key type specific information that a collection implementation relies on. That is, one passes the information how to compare element keys, how to test equality of element keys, type checking relevant information, etc. Which type of information needs to be passed depends on the respective collection interface.

17.2.2 Iterators

Iterators, as defined in this specification, are more than just simple “pointing devices.”

Iterator hierarchy

The service defines a hierarchy of iterators which parallels the collection hierarchy.

The top level iterator is generic in the sense that it allows iteration over all collections, independent of the collection type because it is supported by all collection types. The ordered iterator adds some capabilities useful for all kinds of ordered collections. Iterators further down in the hierarchy add operations exploiting the capabilities of the corresponding collection type Not. Each iterator type is supported by each collection type. For example, a KeyIterator is supported only by collection interfaces derived from KeyCollection.

Iterators are tightly intertwined with collections. An iterator cannot exit independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this, and only this, collection.

Generic and iterator centric programming

Iterators on the one hand are pointer abstractions in the sense of simple pointing devices. They offer the basic capabilities you can expect from a pointer abstraction. One can reset an iterator to a start position for iteration and move or position it in different ways depending on the iterator type.

There are essentially two reasons to embellish an iterator with more capabilities.

1. To support the processing of very large collections to allow for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in “fine grains” via an iterator passed to a client.

2. To enrich the iterator with more capabilities is to strengthen the support for the generic programming model as introduced with ANSI STL to the C++ world.

One can retrieve, replace, remove, and add elements via an iterator. One can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore, an iterator can have a const designation which is set when created. A const iterator can be used for access only.

The reverse iterator semantics is supported. No extra interfaces are specified to support this but a reverse designation is set at creation time. An ordered iterator for which the reverse designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, combined operations and bulk operations are offered.

- Combined operations are combinations of simple iterator operations often used in loops.
- Bulk operations support retrieving, replacing, and adding many elements within one operation.

Managed Iterators

All iterators are managed in the sense that iterators never become undefined; therefore, they do not lead to undefined behavior. Common behavior of iterators in class libraries today is that iterators become undefined when the collection content is changed. For example, if an element is added the side effect on iterators of the collection is unknown. Iterators do not “know” whether they are still pointing to the same element as before, still pointing to an element at all, or pointing “outside” the collection. One cannot even test the state. This is considered unacceptable behavior in a distributed environment.

The iterator model used in this specification is a managed iterator. Managed iterators are “robust” to modifications of the collection. A managed iterator is always in one of the following defined testable states:

- *valid* (pointing to an element of the collection)
- *invalid* (pointing to nothing; comparable to a NULL pointer)
- *in-between* (not pointing to an element, but still "remembering" enough state to be valid for most operations on it).

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum within the collection. There is nothing the managed iterator can point to. Nevertheless, managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a set_to_next_element() for example) without resetting it first. For more information, see "The Managed Iterator Model" on page 17-84.

17.2.3 Function Interfaces

The Object Collection service specifies function interfaces used to pass user-defined information to the collection implementation (either at creation time or as parameters of operations). The most important is the Operations interface discussed in more detail below.

Collectible Elements and Type Safety

Collections are foundation classes used in a broad range of applications. They have to be able to collect elements of arbitrary type and support keys of arbitrary type. Instances of collections are usually homogenous collections in the sense that all elements have the same element type.

Because there is no template support in CORBA IDL today, the requirement "collecting elements of arbitrary type" is met by defining the element type and the key type as a CORBA any. In doing so, compile time type checking for element and key type is impossible.

As collections are often used as homogenous collections, dynamic type checking is enabled by passing relevant information to the collection at creation time. This is done by specialization of the function interface Operations. This interface defines attributes element_type and key_type as well as defines operations check_element_type() and check_key_type() which have to be implemented by the user. Implementations may range from "no type checking at all," "type code match," "checking an interface to be supported," up to "checking constraints in addition to a simple type code checking." Using the Operations interface allows user-defined customization of the dynamic type checking.

Collectible Elements and the Operations Interface

The function interface Operations is used to pass a number of other user-defined element type specific information to the collection implementation.

The type checking of relevant information is one sample.

Depending on the properties represented by a collection interface, a respective implementation relies on some element type specific or key type specific information passed to it. For example, one has to pass the information "element comparison" to implement a SortedSet or "key equality" to guarantee uniqueness of keys in a KeySet. The Operations interface is used to pass this information.

The third use of this interface is to pass element or key type specific information that the different categories of implementations rely on. For example, tree-like implementations for a KeySet rely on the "key comparison" information and hashing based implementations rely on the information how to hash key values. This information is passed via the Operations interface.

A user has to customize the Operations interface and to implement the appropriate operations dependent on the collection interface to be used. An instance of the specialized Operations interface is passed at collection creation time to the collection implementation.

Collectible Elements of Key Collections

Key collections offer associative access to collection elements via a key. A key is computed from the element value and is user-defined element type specific information to be passed to a collection. The Operations interface has an operation key() which returns the user-defined key of a given element.

For a specific element type, a user has to implement the element type specific key() operation in an interface derived from Operations. The key type is a CORBA any. Again this is designed to accommodate generality. Computable keys reflect the data base view on elements of key collections as "keyed elements" where a key is a component of a tuple or is "composed" from several components of a tuple.

17.2.4 List of Interfaces Defined

The Object Collection service offers the following interfaces:

Abstract interfaces representing collection properties and their combinations

- Collection
- OrderedCollection
- KeyCollection
- EqualityCollection
- SortedCollection
- SequentialCollection
- EqualitySequentialCollection
- EqualityKeyCollection
- KeySortedCollection
- EqualitySortedCollection
- EqualityKeySortedCollection

Concrete collections and their factories

- CollectionFactory, CollectionFactories
- KeySet, KeySetFactory
- KeyBag, KeyBagFactory
- Map, MapFactory
- Relation, RelationFactory
- Set, SetFactory
- Bag, BagFactory
- KeySortedSet, KeySortedSetFactory
- KeySortedBag, KeySortedBagFactory
- SortedMap, SortedMapFactory
- SortedRelation, SortedRelationFactory
- SortedSet, SortedSetFactory
- SortedBag, SortedBagFactory
- Sequence, SequenceFactory
- EqualitySequence, EqualitySequenceFactory
- Heap, HeapFactory

Restricted access collections and their factories

- RestrictedAccessCollection, RACollectionFactory
- Stack, StackFactory
- Queue, QueueFactory
- Deque, DequeFactory
- PriorityQueue, PriorityFactory

Iterator interfaces

- Iterator
- OrderedIterator
- SequentialIterator
- SortedIterator
- KeyIterator
- EqualityIterator
- EqualityKeyIterator

- KeySortedIterator
- EqualitySortedIterator
- EqualitySequentialIterator
- EqualityKeySortedIterator

Function interfaces

- Operations
- Command
- Comparator

17.3 Combined Collections

The overview introduced *properties* and listed the meaningful combinations of these properties that result in consistently defined collection interfaces forming a differentiated offering. In the following sections, the semantics of each combination will be described in more detail and demonstrated by an example.

17.3.1 Combined Collections Usage Samples

Bag, SortedBag

A Bag is an unordered collection of zero or more elements with no key. Multiple elements are supported. As element equality is supported, operations which require the capability "test of element equality" (e.g., test on containment) can be offered.

Example: The implementation of a text file compression algorithm. The algorithm finds the most frequently occurring words in sample files. During compression, the words with a high frequency are replaced by a code (for example, an escape character followed by a one character code). During re-installation of files, codes are replaced by the respective words.

Several types of collections may be used in this context. A Bag can be used during the analysis of the sample text files to collect isolated words. After the analysis phase you may ask for the number of occurrences for each word to construct a structure with the 255 words with the highest word counts. A Bag offers an operation for this, you do not have to "count by hand," which is less efficient. To find the 255 words with the highest word count, a SortedRelation is the appropriate structure (see "Relation, SortedRelation" on page 17-13). Finally, a Map may be used to maintain a mapping of words to codes and vice versa. (See "Map, SortedMap" on page 17-12).

A SortedBag (as compared to a Bag) exposes and maintains a sorted order of the elements based on a user-defined element comparison. Maintained elements in a sorted order makes sense when printing or displaying the collection content in sorted order.

EqualitySequence

An EqualitySequence is an ordered collection of elements with no key. There is a first and a last element. Each element, except the last one, has a next element and each element, except the first one, has a previous element. As element equality is supported, all operations that rely on the capability "test on element equality" can be offered, for example, locating an element or test for containment.

Example: An application that arranges wagons to a train. The order of the wagons is important. The trailcar has to be the first wagon, the first class wagons are arranged right behind the trailcar, the restaurant has to be arranged right after the first class and before the second class wagons, and so on. To check whether the wagon has the correct capacity, you may want to ask: "How many open-plan carriages are in the train?" or "Is there a bistro in the train already?"

Heap

A Heap is an unordered collection of zero or more elements without a key. Multiple elements are supported. No element equality is supported.

Example: A "trash can" on a desktop which memorizes all objects moved to the trashcan as long as it is not emptied. Whenever you move an object to the trashcan it is added to the heap. Sometimes you move an object accidentally to the trashcan. In that case, you iterate in some order through the trashcan to find the object - not using a test on element equality. When you find it, you remove it from the trashcan. Sometimes you empty the trashcan and remove all objects from the trashcan.

KeyBag, KeySortedBag

A KeyBag is an unordered collection of zero or more elements that have a key. Multiple keys are supported. As no element equality is assumed, operations such as "test on collection equality" or "set theoretical operation" are not offered.

A KeySortedBag is sorted by key. In addition to the operations supported for a KeyBag, all operations related to ordering are offered. For example, operations exploiting the ordering such as "set_to_previous / set_to_next" and "access via position" are supported.

A license server maintaining floating licenses on a network may be implemented using a KeyBag to maintain the licenses in use. The key may be the LicenseId and additional element data may be, for example, the user who requested the license. As usual, more than one floating license is available per product; therefore, many licenses for the same product may be in use. A LicenseId may occur more than once. A user may request a license multiple times, it may also occur that the same LicenseId with the same user occurs multiple times. If a user of the product requests and receives the license, the LicenseId, together with the request data, is added to the licenses in use. If the license is released, it is deleted from the Bag of licenses in use. Sometimes you may want to ask for the number of licenses of a product in use, that is ask for the number of the licenses in use with a given LicenseId.

Access to licenses in use is via the key LicenseId. This sample application does not require operations such as testing two collections for equality or set theoretical operations on collections. It is not exploiting element equality; therefore, it can use a KeyBag instead of a Relation (which would force the user to define element equality).

If you want to list the licenses in use with the users holding the licenses sorted by LicenseId, you could make use of a KeySortedBag instead of a KeyBag.

KeySet, KeySortedSet

A KeySet is an unordered collection of zero or more elements that have a key. Keys must be unique. Defined element equality is not assumed; therefore, operations and semantics which require the capability “element equality test" are not offered.

A KeySortedSet is sorted by key. In addition to the operations supported for a KeySet, all operations related to ordering are offered. For example, operations exploiting the ordering, such as “set_to_previous / set_to_next” and “access via position” are supported.

Example: A program that keeps track of cancelled credit card numbers and the individuals to whom they are issued. Each card number occurs only once and the collection is sorted by card number. When a merchant enters a customer’s card number into the point-of-sales terminal, the collection is checked to determine whether the card number is listed in the collection of cancelled cards. If it is found, the name of the individual is shown and the merchant is given directions for contacting the card company. If the card number is not found, the transaction can proceed because the card is valid. A list of cancelled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Map, SortedMap

A Map is an unordered collection of zero or more elements that have a key. Keys must be unique. As defined, element equality is assumed access via the element value and all operations which need to test on element equality, such as a test on containment for an element, test for equality, and set theoretical operations can be offered for maps.

A SortedMap is sorted by key. In addition to the operations supported for a Map, all operations related to ordering are offered. For example, operations exploiting the ordering like “set_to_previous / set_to_next” and “access via position” are supported.

Example: Maintaining nicknames for your mailing facility. The key is the nickname. Mailing information includes address, first name, last name, etc. Nicknames are unique; therefore, adding a nickname/mailing inforation entry with a nickname that is already available should fail, if the mailing information to be added is different from the available information. If it is exactly the same information, it should just be ignored. You may define more than one nickname for the same person; therefore, the same element data may be stored with different keys. If you want to update address information for a given nickname, use the replace_element_with_key() operation. To create a new nickname file from two existing files, use a union operation which assumes element equality to be defined.

Relation, SortedRelation

A Relation is an unordered collection of zero or more elements with a key. Multiple keys are supported. As defined element equality is assumed, test for equality of two collections is offered as well as the set theoretical operations.

A SortedRelation is sorted by key. In addition to the operations supported for a Relation, all operations related to ordering are offered. For example, operations that exploit ordering such as "set_to_previous / set_to_next" and "access via position" are supported.

A SortedRelation may be used in the text file compression algorithm mentioned previously in the Bag, Sorted Bag example to find the 255 words with the highest frequency. The key is the word count and the additional element data is the word. As words may have equal counts, multiple keys have to be supported. The ordering with respect to the key is used to find the 255 highest keys.

Set, SortedSet

A set is an unordered collection of zero or more elements without a key. Element equality is supported; therefore, operations that require the capability "test on element equality" such as intersection or union can be offered.

A SortedSet is sorted with respect to a user-defined element comparison. In addition to the operations supported for a Set, all operations related to ordering are offered. For example, operations that exploit ordering such as "set_to_previous / set_to_next" and "access via position" are supported.

Example: A program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is added to the set. The set does not allow an item to be added if it already is present in the collection; this ensures that a customer does not get two samples of a single product.

Sequence

A Sequence is an ordered collection of elements without a key. There is a first and a last element. Each element (except the last one) has a next element and each element (except the first one) has a previous element. No element equality is supported; therefore, multiples may occur and access to elements via the element value is not possible. Access to elements is possible via position/index.

Example: A music editor. The Sequence is used to maintain tokens representing the recognized notes. The order of the notes is obviously important for further processing of the melody. A note may occur more than once. During editing, notes are accessed by position and are removed, added, or replaced at a given position. To print the result, you may iterate over the sequence and print note by note.

A Sequence may also be used to represent how a book is constructed from diverse documents. It is obvious that ordering is important. It may be the case that a specific document is used multiple times within the same book (for example, a specific graphic). Reading the book, you may want to access a specific document by position.

17.4 Restricted Access Collections

17.4.1 Restricted Access Collections Usage Samples

Deque

A double ended queue may be considered as a sequence with restricted access. It is an ordered collection of elements without a key and no element equality. As there is no element equality, an element value may occur multiple times. There is a first and a last element. You can only add an element as first or last element and only remove the first or the last element from the Deque.

A Deque may be used in the implementation of a pattern matching algorithm where patterns are expressed as regular expressions. Such an algorithm can be described as a non-deterministic finite state machine constructed from the regular expression. The implementation of the regular-pattern matching machine may use a deque to keep track of the states under consideration. Processing a null state requires a stack-like data structure - one of two things to be done is postponed and put at the front of the not being postponed forever list. Processing the other states requires a queue-like data structure, since you do not want to examine a state for the next given character until you are finished with the current character. Combining the two characteristics results in a Deque.

PriorityQueue

A PriorityQueue may be considered as a KeySortedBag with restricted access. It is an ordered collection with zero or more elements. Multiple key values are supported. As no element equality is defined, multiple element values may occur. Access to elements is via key only and sorting is maintained by key. Accessing a PriorityQueue is restricted. You can add an element relative to the ordering relation defined for keys and remove only the first element (e.g., the one with highest priority).

PriorityQueues may be used for implementing a printer queue. A print job's priority may depend on the number of pages, time of queuing, and other characteristics. This priority is the key of the print job. When a user adds a print job it is added relative to its priority. The printer daemon always removes the job with the highest priority from the queue.

PriorityQueues also may be used as special queues in workflow management to prioritize work items.

Queue

A queue may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (enque) an element as last element and only remove (deque) the first element from the Queue. That is, a queue exposes FIFO behavior.

You would use a queue in tree traversal to implement a breadth first search algorithm.

Queues may be used for the implementation of all kinds of buffered communication where it is important that the receiving side handles messages in the same order as they were sent. Queues may be used in workflow management environments where queues collect messages waiting for processing.

Stack

A Stack may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (push) an element as last element (at the top) and only remove (pop) the last element from the Stack (from the top). That is, a Stack exposes LIFO behavior. The classical application for a stack is the simulation of a calculator with Reverse Polish Notation. The calculator engine may get an arithmetic expression. Parsing the expression operands are pushed on to the stack. When an operator is encountered, the appropriate number of operands is popped off the stack, the operation performed, and the result pushed on the stack.

A Stack also may be used in the implementation of a window manager to maintain the order in which the windows are superimposed.

17.5 The CosCollection Module

17.5.1 Interface Hierarchies

Collection Interface Hierarchies

The collection interfaces of the Collection Services are organized in two separate hierarchies, as shown in Figure 17-1 on page 17-17 and Figure 17-2 on page 17-17. The inner nodes of the hierarchy may be thought of as abstract views. They represent the basic properties and their combinations. Leaf nodes may be thought of as concrete interfaces for which implementations are provided and from which instances can be created via a collection factory. The organization of the interfaces as a hierarchy enables reuse and the polymorphic usage of the collections from typed languages such as C++.

Each abstract view is defined in terms of operations and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. This view allows adding elements to and iterating over the collection.

In addition to the common collection operations, collections whose elements define equality or key equality provide operations for locating and retrieving elements by a given element or key value.

Ordered collections provide the notion of well-defined explicit positioning of elements, either by element key ordering relation or by positional element access.

Sorted collections provide no further operations, but introduce a new semantics; namely, that their elements are sorted by element or key value. These properties are combined through multiple inheritance.

The fourth property, uniqueness/multiplicity of elements and keys, is not represented by a separate abstract view for combination with other properties. This was done to reduce the complexity of the hierarchy. Instead, operations related to multiplicity are provided in the base interface from which the interface specializations with multiplicity are derived.

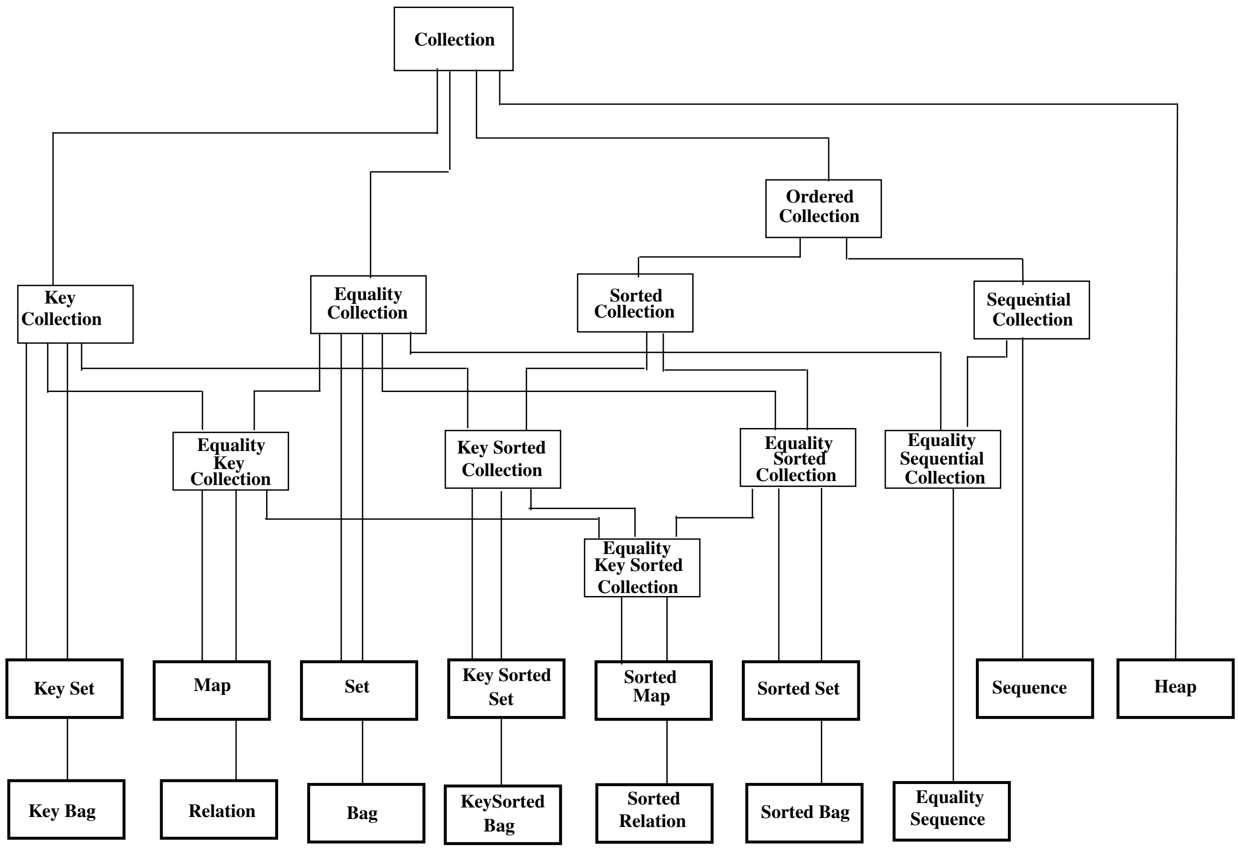


Figure 17-1 Collections Interfaces Hierarchy

The restricted access collections form their own hierarchy as shown in Figure 17-2 on page 17-17. This abstract view defines the operations that all restricted access collections have in common.

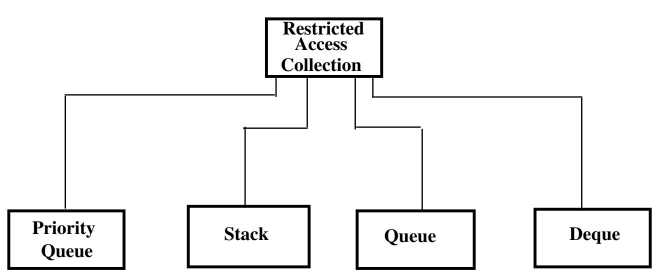


Figure 17-2 Restricted Access Collections Interface Hierarchy

Iterator Hierarchy

The iterator interface hierarchy parallels the Collection interface hierarchy shown in Figure 17-3 on page 17-18. The defined interfaces support the fine-grain processing of very large collections via an iterator only and support a generic programming model similar to what was introduced with ANSI STL to the C++ world. Concepts like constness of iterators, reverse iterators, bulk and combined operations are offered to strengthen the support for the generic programming model.

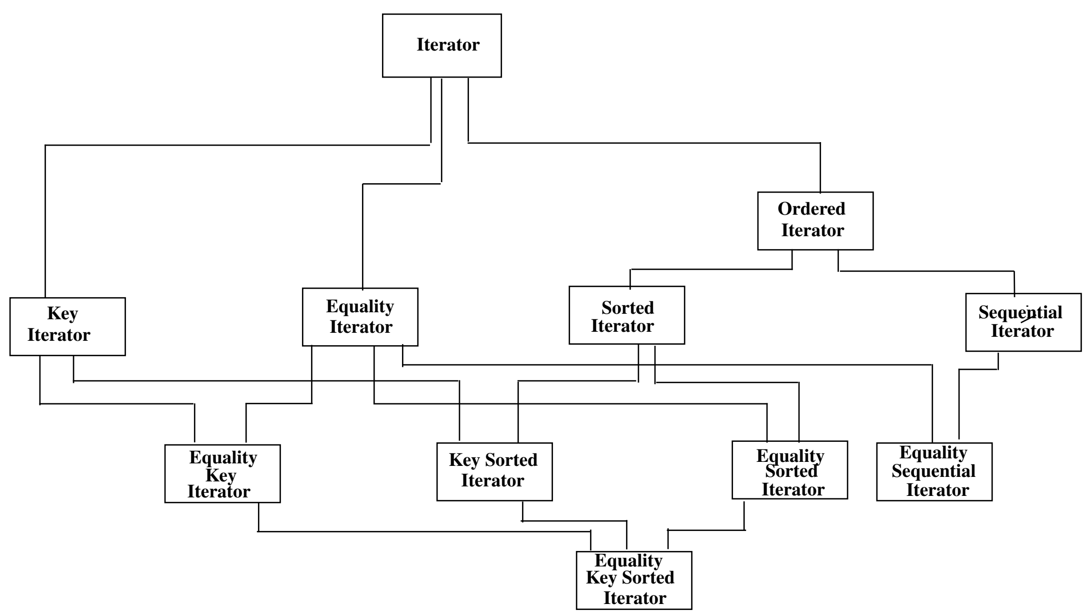


Figure 17-3 Iterator Interface Hierarchy

The top level Iterator interface represents a generic iterator that can be used for iteration over and manipulation of all collections independent of their type. The top level iterator allows you to add, retrieve, replace, and remove elements. There are operations to clone, assign, and test iterators for equality. There are tests on the iterator state and you can check whether an iterator is *const*, created for a given collection, or created for the same collection as another iterator.

The OrderedIterator interface adds those operations which are useful on collections with an explicit notion of ordering (all those collections inheriting from the OrderedCollection interface). An ordered iterator can be moved forward and backward, set to a position, and its position can be computed. Only ordered iterators can be used with "reverse" semantics. The SequentialIterator is used with sequentially ordered collections where it is possible to add elements at a user-defined position so that the iterator offers the capability to add elements relative to its position.

The KeyIterator and EqualityIterator interface add operations for positioning an iterator by key or element value. The sorted versions of these interfaces add respective backward movements and the capability to define lower and upper bounds in sorted collections.

An iterator is always created for a collection using the collection as iterator factory. Each iterator type is supported by each collection type. The Iterators and the Collections that are supported by all interfaces derived from those collections are listed in Table 17-2 on page 17-19.

Table 17-2 Iterators and Collections

	Supported by all interfaces derived from:
Iterator	**Collection**
OrderedIterator	OrderedCollection
SequentialIterator	SequentialCollection
EqualitySequentialIterator	EqualitySequentialCollection
KeyIterator	KeyCollection
EqualityIterator	EqualityCollection
EqualityKeyIterator	EqualityKeyCollection
SortedIterator	SortedCollection
KeySortedIterator	KeySortedCollection
EqualitySortedIterator	EqualitySortedCollection
EqualityKeySortedIterator	EqualityKeySortedCollection

17.5.2 Exceptions and Type Definitions

The following exceptions are used by the subsequently defined interfaces.

```
module CosCollection {
// Type definitions
typedef sequence<any> AnySequence;
typedef string Istring;
struct NVPair {Istring name; any value;};
typedef sequence<NVPair> ParameterList;


//      Exceptions
exception EmptyCollection{};
```

```
exception PositionInvalid{};
enum IteratorInvalidReason {is_invalid, is_not_for_collection, is_const};
exception IteratorInvalid {IteratorInvalidReason why;};
exception IteratorInBetween{};
enum ElementInvalidReason {element_type_invalid, positioning_property_invalid,
element_exists};
exception ElementInvalid {ElementInvalidReason why;};
exception KeyInvalid {};
exception ParameterInvalid {unsigned long which; Istring why;};
```

AnySequence

A type definition for a sequence of values of type any used in bulk operations.

Istring

A type definition used as place holder for a future IDL internationalized string data type.

ParameterList

A sequence of name-value pairs of type NVPair and used as a generic parameter list in a generic collection creation operation.

EmptyCollection

Raised when an operation to remove an element is invoked on an empty collection.

PositionInvalid

Raised when an operation on an ordered collection passes a position out of the allowed range, that is less than 1 or greater than the number of elements in the collections.

IteratorInvalid

Raised when an operation uses an iterator pointing to nothing, that is, using an *invalid* iterator (in_valid) or when an operation uses an iterator which was not created for the collection (is_not_for_collection) or if one tries to modify a collection via an iterator that is created with const designation (is_const).

IteratorInBetween

Raised when an operation uses an iterator in a way that does not allow the state *in-between* such as all "..._at" operations.

ElementInvalid

Raised when one of the operations passes an element that is for one of several reasons invalid. It is raised

- when the element is not of the expected element type (element_type_invalid).
- if one tries to replace an element by another element changing the positioning property (positioning_property_invalid).
- when an element is added to a Map and the key already exists (element_exists).

KeyInvalid

Raised when one of the operations passes a key that is not of the expected type.

Paramete rInvalid

Raised when a parameter passed to the generic collection creation operation of the generic CollectionFactory is invalid.

17.5.3 Abstract Collection Interfaces

The Collection Interface

The Collection interface represents the most abstract view of a collection. Operations defined in this top level interface can be supported by all collection interfaces in the hierarchy. Each concrete collection interface offers the appropriate operation semantics dependent on the collection properties. It defines operations for:

- adding elements
- removing elements
- replacing elements
- retrieving elements
- inquiring collection information
- creating iterators

```
// Collection
interface Iterator;
interface Command;

interface Collection {

// element type information
readonly attribute CORBA::TypeCode element_type;

// adding elements
```

```
boolean add_element (in any element) raises (ElementInvalid);
boolean add_element_set_iterator (in any element, in Iterator where) raises (IteratorInvalid,
ElementInvalid);
void add_all_from (in Collection collector) raises (ElementInvalid);

// removing elements
void remove_element_at (in Iterator where) raises (IteratorInvalid, IteratorInBetween);
unsigned long remove_all ();

// replacing elements
void replace_element_at (in Iterator where, in any element) raises(IteratorInvalid, IteratorInBetween,
ElementInvalid);


// retrieving elements
boolean retrieve_element_at (in Iterator where, out any element) raises (IteratorInvalid,
IteratorInBetween);

// iterating over the collection
boolean all_elements_do (in Command what) ;

// inquiring collection information
unsigned long number_of_elements ();
boolean is_empty ();

// destroying collection
void destroy();

// creating iterators
Iterator create_iterator (in boolean read_only);
};
```

Type checking information

readonly attribute CORBA::TypeCode element_type;

Specifies the element type expected in the collection. See also "The Operations Interface" on page 17-117.

Adding elements

boolean add_element (in any element) raises (ElementInvalid);

Description

Adds an element to the collection. The exact semantics of the add operations depends on the properties of the concrete interface derived from the Collection that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception ElementInvalid is raised.

Return value

Returns true if the element is added.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);

Description

Adds an element to the collection and sets the iterator to the added element. The exact semantics of the add operations depends on the properties of the concrete interface derived from the Collection that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception ElementInvalid is raised.

Return value

Returns true if the element is added.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All other iterators keep their state.

```
void add_all_from (in Collection elements) raises (ElementInvalid);
```

Adds all elements of the given collection to this collection. The elements are added in the iteration order of the given collection and consistent with the semantics of the add operation. Essentially, this operation is a sequence of add operations.

Removing elements

```
void remove_element_at (in Iterator where) raises(IteratorInvalid);
```

Description

Removes the element pointed to by the given iterator. The given iterator is set to *in-between*.

Exceptions

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

Iterators pointing to the removed element go *in-between*. Iterators which do not point to the removed element keep their state.

```
unsigned long void remove_all();
```

Description

Removes all elements from the collection.

Return value

Returns the number of elements removed.

Side effects

Iterators pointing to removed elements go *in-between*. All other iterators keep their state.

Replacing elements

void replace_element_at (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid)

Description

Replaces the element pointed to by the iterator by the given element. The given element must have the same positioning property as the replaced element.

- For collections organized according to element properties such as ordering relation, the replace operation must not change this element property.
- For key collections, the new key must be equal to the key replaced.
- For non-key collections with element equality, the new element must be equal to the replaced element as defined by the element equality relation.

Sequential collections have a user-defined positioning property and heaps do not have positioning properties. Element values in sequences and heaps can be replaced freely.

Exceptions

The given element must not change the positioning property; otherwise, the exception ElementInvalid is raised.

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Retrieving elements

boolean retrieve_element_at (in Iterator where, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed to by the given iterator and returns it via the output parameter element.

Return value

Returns true if an element is retrieved.

Exceptions

The given iterator must belong to the collection and must point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the any. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

Iterating over a collection

boolean all_elements_do (in Command what);

Description

Calls the "do_on()" operation of the given Command for each element of the collection until the "do_on()" operation returns false. The elements are visited in iteration order (see "The Command and Comparator Interface" on page 17-121).

- The "do_on()" operation must not remove elements from or add elements to the collection.
- The "do_on()" operation must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return value

Returns true if the "do_on()" operation returns true for each element it is applied to.

Inquiring collection information

The collection operations do have preconditions which when violated raise exceptions. There are operations for testing those preconditions to enable the user to avoid raising exceptions.

unsigned long number_of_elements ();

Return value

Returns the number of elements contained in the collection.

boolean is_empty ();

Return value

Returns true if the collection is empty.

Destroying a collection

void destroy();

Description

Destroys the collection. This includes:

- removing all elements from the collection
- destroying all iterators created for this collection
- destroying the instance of Operations passed at creation time to the collection implementation.

Note – Removing elements in case of objects means removing object references, not destroying the collected objects.

Object references to iterators of the collections become invalid.

Creating iterators

Iterator create_iterator (in boolean read_only);

Creates and returns an iterator instance for this collection. The type of iterator that is created depends on the interface type of this collection. The following table describes the type of iterator that is created for the type of concrete collection.

Table 17-3 Collection interfaces and the iterator interfaces supported

Ordered	**Collection Interfaces**	**Supported Iterator Interface**
	Bag	EqualityIterator
yes	SortedBag	EqualitySortedIterator
yes	EqualitySequence	EqualitySequentialIterator
	Heap	Iterator
	KeyBag	KeyIterator
yes	KeySortedBag	KeySortedIterator
	KeySet	KeyIterator
yes	KeySortedSet	KeySortedIterator
	Map	EqualityKeyIterator
yes	SortedMap	EqualityKeySortedIterator
	Relation	EqualityKeyIterator
yes	Sequence	SequentialIterator

Table 17-3 Collection interfaces and the iterator interfaces supported

yes	SortedRelation	EqualityKeySortedIterator
	Set	EqualityIterator
yes	SortedSet	EqualitySortedIterator
yes	Sequence	SequentialIterator

After creation, the iterator is initialized with the state *invalid,* that is, "pointing to nothing."

If the given parameter read_only is true, the iterator is created with const designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception IteratorInvalid).

Note – Collections serve as factories for *their* iterator instances. An iterator is created in the same address space as the collection for which it is created. An iterator instance can only point to elements of the collection for which it was created.

The OrderedCollection Interface

```
interface OrderedIterator;
// OrderedCollection
interface OrderedCollection: Collection {

// removing elements
void remove_element_at_position (in unsigned long position) raises (PositionInvalid);
void remove_first_element () raises (EmptyCollection);
void remove_last_element () raises (EmptyCollection);

// retrieving elements
boolean retrieve_element_at_position (in unsigned long position, out any element) raises
(PositionInvalid);
boolean retrieve_first_element (out any element) raises (EmptyCollection);
boolean retrieve_last_element (out any element) raises (EmptyCollection);
// creating iterators
OrderedIterator create_ordered_iterator(in boolean read_only, in boolean reverse_iteration);
};
```

Ordered collections expose the ordering of elements in their interfaces. Elements can be accessed at a position and forward and backward movements are possible (i.e., ordered collection can support ordered iterators). Ordering can be implicitly defined via the ordering relationship of the elements or keys (as in sorted collections) or ordering can be user-controlled (as in sequential collections).

In addition to those inherited from the Collection Interface, which all ordered collections have in common, the OrderedCollection interface provides operations for

- removing elements,
- retrieving elements, and
- creating ordered iterators.

Removing elements

void remove_element_at_position (in unsigned long position) raises (PositionInvalid);

Description

Removes the element from the collection at a given position. The first element of the collection has position 1.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception PositionInvalid is raised. A position is valid if it is greater than or equal to 1 and less than or equal to number_of_elements().

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

void remove_first_element () raises (EmptyCollection);

Description

Removes the first element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

void remove_last_element () raises (EmptyCollection);

Description

Removes the last element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

Retrieving elements

boolean retrieve_element_at_position (in unsigned long position, out any element) raises (PositionInvalid);

Description

Retrieves the element at the given position in the collection and returns it via the output parameter element. Position 1 specifies the first element.

Return value

Returns true if an element is retrieved.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception PositionInvalid is raised.

boolean retrieve_first_element (out any element) raises (EmptyCollection);

Description

Retrieves the first element in the collection and returns it via the output parameter element.

Return value

Returns true if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

boolean retrieve_last_element (out any element) raises (EmptyCollection);

Description

Retrieves the last element in the collection and returns it via the output parameter element.

Return value

Returns true if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

Creating iterators

OrderedIterator create_ordered_iterator (in boolean read_only, in boolean reverse_iteration);

Description

Creates and returns an ordered iterator instance for this collection.

Which type of ordered iterator actually is created depends on the interface type of this collection. Table 17-1 on page 17-4 describes which type of ordered iterator is created for which type of concrete ordered collection.

After creation, the iterator is initialized with the state invalid, that is, "pointing to nothing."

Exceptions

If the given parameter read_only is true, the iterator is created with const designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception IteratorInvalid).

Side effects

If the given parameter reverse_iteration is true, the iterator is created with reverse iteration semantics. Only ordered iterators can be created with reverse semantics.

The SequentialCollection Interface

interface Comparator;

interface SequentialCollection: OrderedCollection {

// adding elements

void add_element_as_first (in any element) raises (ElementInvalid);

void add_element_as_first_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_last (in any element) raises (ElementInvalid);

void add_element_as_last_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_next (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_previous (in any element, in Iterator where) raises (ElementInvalid,IteratorInvalid);

void add_element_at_position (in unsigned long position, in any element) raises(PositionInvalid, ElementInvalid);

void add_element_at_position_set_iterator (in unsigned long position, in any element, in Iterator where) raises (PositionInvalid, ElementInvalid, IteratorInvalid);

// replacing elements

void replace_element_at_position (in unsigned long position, in any element) raises (PositionInvalid, ElementInvalid);

void replace_first_element (in any element) raises (ElementInvalid, EmptyCollection);

void replace_last_element (in any element) raises (ElementInvalid, EmptyCollection);

// reordering elements

void sort (in Comparator comparison);

void reverse();

};

Sequential collections expose user-controlled sequential ordering. Determine where elements are added by comparing to sorted collections where the “where an element is added“ is determined implicitly by the defined element or key comparison.

The SequentialCollection interface adds all those operations to the OrderedCollection interface. “The SequentialCollection Interface” on page 17-31 describes operators that are unique for positional element access for

- adding elements,
- replacing elements, and
- re-ordering elements.

Adding elements

void add_element_as_first (in any element) raises (ElementInvalid);

Description

Adds the element to the collection as the first element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

void add_element_as_first_set_iterator (in any element, in Iterator where) raises (ElementInvalid,IteratorInvalid);

Description

Adds the element to the collection as the first element in sequential order and sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

void add_element_as_last (in any element) raises (ElementInvalid);

Description

Adds the element to the collection as the last element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

void add_element_as_last_set_iterator (in any element, in Iterator where) raises (ElementInvalid,IteratorInvalid);

Description

Adds the element to the collection as the last element in sequential order. Sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All other iterators keep their state.

void add_element_as_next(in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Adds the element to the collection after the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between*, the element is added before the iterator's "potential next" element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection and be valid; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

void add_element_as_previous (in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);

Description

Adds the element to the collection as the element previous to the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between,* the element is added after the iterator's "potential previous" element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

void add_element_at_position (in unsigned long position, in any element) raises(PositionInvalid, ElementInvalid);

Description

Adds the element at the given position to the collection. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to number_of_elements() +1); otherwise, the exception PositionInvalid is raised.

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

void add_element_at_position_set_iterator (in unsigned long position, in any element, in Iterator where) raises (PositionInvalid, ElementInvalid IteratorInvalid);

Description

Adds the element at the given position to the collection and sets the iterator to the added element. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to number_of_elements() +1); otherwise, the exception PositionInvalid is raised.

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

Replacing elements

void replace_element_at_position (in unsigned long position, in any element) raises (PositionInvalid, ElementInvalid);

Description

Replaces the element at a given position with the given element. The given position must be valid (i.e., greater than or equal to 1 and less than or equal to number_of_elements()).

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

void replace_first_element (in any element) raises (ElementInvalid, EmptyCollection);

Description

Replaces the first element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

void replace_last_element (in any element) raises (ElementInvalid, EmptyCollection);

Description

Replaces the last element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

Re-ordering elements

void sort (in Comparator comparison);

Description

Sorts the collection so that the elements occur in ascending order. The relation of two elements is defined by the "compare" method, which a user provides when implementing an interface derived from Comparator. See "The Command and Comparator Interface" on page 17-121.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

void reverse ();

Description

Orders elements in reverse order.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

The SortedCollection Interface

interface SortedCollection: OrderedCollection{};

Sorted collections currently do not provide further operations but define a more specific behavior; namely, that the elements or their keys are sorted with respect to a user-defined element or key compare. See "The OrderedCollection Interface" on page 17-28.

The EqualityCollection Interface

interface EqualityCollection: Collection {

// testing element containment

boolean contains_element (in any element) raises(ElementInvalid);

boolean contains_all_from (in Collection collector) raises(ElementInvalid);

// adding elements

boolean locate_or_add_element (in any element) raises (ElementInvalid);

boolean locate_or_add_element_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

// locating elements

boolean locate_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

boolean locate_next_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

boolean locate_next_different_element (in Iterator where) raises (IteratorInvalid, IteratorInBetween);

// removing elements

```
boolean remove_element (in any element) raises (ElementInvalid);
unsigned long remove_all_occurrences (in any element) raises (ElementInvalid);

// inquiring collection information
unsigned long number_of_different_elements ();
unsigned long number_of_occurrences (in any element) raises(ElementInvalid);
};
```

Collections whose elements define equality introduce operations which exploit the defined element equality. These operations are for finding elements by element value (and adding if not found), for testing containment of a given element, and inquiring the collection about how many elements of a given value were collected.

Testing element containment

boolean contains_element (in any element) raises (ElementInvalid);

Return value

Returns true if the collection contains an element equal to the given element.

Exceptions

The given elements must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean contains_all_from (in Collection collector) raises (ElementInvalid);

Return value

Returns true if all the elements of the given collection are contained in the collection. The definition of containment is given in “contains_element.”

Exceptions

The elements in the given collection must be of the expected type; otherwise, the exception ElementInvalid is raised.

Adding elements

boolean locate_or_add_element (in any element) raises (ElementInvalid);

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in add.

Return value

Returns true if the element was found.

Returns false if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

boolean locate_or_add_element_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in add. The iterator is set to the found or added element.

Return value

Returns true if the element was found.

Returns false if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All other iterators keep their state.

Locating elements

boolean locate_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns true if an element is found.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

boolean locate_next_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the located element. The iterator is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited. If the iterator is in the state *in-between,* locating is started at the iterator's "potential next" element.

Return value

Returns true if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception IteratorInvalid is raised.

boolean locate_next_different_element (in Iterator where) raises (IteratorInvalid, IteratorInBetween);

Description

Locates the next element in iteration order that is different from the element pointed to by the given iterator. If no more elements are left to be visited, the given iterator will no longer be valid.

Return value

Returns true if the next different element was found.

Exception

The iterator must belong to the collection and point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Removing elements

boolean remove_element (in any element) raises (ElementInvalid);

Description

Removes an element in the collection that is equal to the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of the given element will be removed.

Return value

Returns true if an element was removed.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

If an element was removed, all iterators pointing to this element go *in-between*.

All other iterators keep their state.

unsigned long remove_all_occurrences (in any element) raises (ElementInvalid);

Description

Removes all elements from the collection that are equal to the given element and returns the number of elements removed.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators pointing to elements removed go *in-between*.

All iterators keep their state.

Inquiring collection information

unsigned long number_of_different_elements ();

Return value

Returns the number of different elements in the collection.

unsigned long number_of_occurrences (in any element) raises (ElementInvalid);

Return value

Returns the number of occurrences of the given element in the collection.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The KeyCollection Interface

```
interface KeyCollection: Collection {
// Key type information
readonly attribute CORBA::TypeCode key_type;

// testing containment
boolean contains_element_with_key (in any key) raises(KeyInvalid);
boolean contains_all_keys_from (in KeyCollection collector) raises(KeyInvalid);

// adding elements
boolean locate_or_add_element_with_key (in any element) raises(ElementInvalid);
boolean locate_or_add_element_with_key_set_iterator (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);

// adding or replacing elements
boolean add_or_replace_element_with_key (in any element) raises(ElementInvalid);
boolean add_or_replace_element_with_key_set_iterator (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);

// removing elements
```

```
boolean remove_element_with_key(in any key) raises(KeyInvalid);
unsigned long remove_all_elements_with_key (in any key) raises(KeyInvalid);

// replacing elements
boolean replace_element_with_key (in any element) raises(ElementInvalid);
boolean replace_element_with_key_set_iterator (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);

// retrieving elements
boolean retrieve_element_with_key (in any key, out any element) raises (KeyInvalid);
// computing the keys
void key (in any element, out any key) raises (ElementInvalid);
void keys (in AnySequence elements, out AnySequence keys) raises (ElementInvalid);

// locating elements
boolean locate_element_with_key (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);
booleanlocate_next_element_with_key(inanykey,inIteratorwhere)raises(KeyInvalid,IteratorInvalid);
boolean locate_next_element_with_different_key (in Iterator where) raises (IteratorInBetween,
IteratorInvalid);

// inquiring collection information
unsigned long number_of_different_keys ();
unsigned long number_of_elements_with_key (in any key) raises(KeyInvalid);
};
```

A KeyCollection is a collection which offers associative access to its elements via a key. All elements of such a collection are keyed elements (i.e., they do have a key which is computed from the element value). How to compute the key from an element value is user-defined. A user specializes the Operations interface and implements the operation key() as desired (see "The Operations Interface" on page 17-117). This information is passed to the collection at creation time.

Type checking information

readonly attribute CORBA::TypeCode key_type;

Specifies the key type expected in the collection. See also "The Operations Interface" on page 17-117.

Testing containment

boolean contains_element_with_key (in any key) raises (KeyInvalid);

Return value

Returns true if the collection contains an element with the same key as the given key.

Exceptions

The given key has to be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean contains_all_keys_from (in KeyCollection collector) raises(KeyInvalid);

Return value

Returns true if all of the keys of the given collection are contained in the collection.

Exceptions

The keys of the given collection have to be of the expected type of this collection; otherwise, the exception KeyInvalid is raised.

Adding elements

boolean locate_or_add_element_with_key (in any element) raises(ElementInvalid);

Description

Locates an element with the same key as the key in the given element. If no such element exists the element is added; otherwise, the collection remains unchanged.

Return value

Returns true if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

boolean locate_or_add_element_with_key_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates an element with the same key as the key in the given element and sets the iterator to the located elements (see locate_element_with_key()). If no such element exists, the element is added and the iterator is set to the element added.

Return value

Returns true if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

boolean add_or_replace_element_with_key (in any element) raises (ElementInvalid);

Description

If the collection contains an element with the key equal to the key in the given element, the element is replaced with the given element; otherwise, the given element is added to the collection.

Return value

Returns true if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators keep their state.

boolean add_or_replace_element_with_key_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

If the collection contains an element with the key equal to the key in the given element, the iterator is set to that element and the element is replaced with the given element; otherwise, the given element is added to the collection, and the iterator set to the added element.

Return value

Returns true if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Side effects

All iterators keep their state.

Removing elements

boolean remove_element_with_key (in any key) raises (KeyInvalid);

Description

Removes an element from the collection with the same key as the given key. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be removed.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

Side effects

If an element was removed, all iterators pointing to the element go *in-between*.

All other iterators keep their state.

unsigned long remove_all_elements_with_key (in any key) raises(KeyInvalid);

Description

Removes all elements from the collection with the same key as the given key.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

Side effects

Iterators pointing to elements removed go *in-between*.

All other iterators keep their state.

Replacing elements

boolean replace_element_with_key (in any element) raises (ElementInvalid);

Description

Replaces an element with the same key as the given element by the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns true if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean replace_element_with_key_set_iterator (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Replaces an element with the same key as the given element by the given element, and sets the iterator to this element. If no such element exists, the iterator is invalidated and the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns true if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

Computing keys

void key (in any element, out any key) raises(ElementInvalid);

Description

Computes the key of the given element and returns it via the output parameter key.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

void keys (in Any Sequence elements, out Any Sequence keys) raises(ElementInvalid);

Description

Computes the keys of the given elements and returns them via the output parameter keys.

Exceptions

The given elements must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

An implementation may rely on the key operation of a user supplied interface derived from Operations. An instance of this interface is passed to a collection at creation time and can be used in the collection implementation.

Locating elements

boolean locate_element_with_key (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_next_element_with_key (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates the next element in iteration order with the key equal to the given key, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the element in the collection. The given iterator is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited. If the iterator is in the *in-between* state, locating starts at the iterator's "potential next" element.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must belong to the collection and must be valid; otherwise, the exception IteratorInvalid is raised.

boolean locate_next_element_with_different_key (in Iterator where) raises(IteratorInvalid, IteratorInBetween)

Description

Locates the next element in the collection in iteration order with a key different from the key of the element pointed to by the given iterator. If no such element exists, the given iterator is no longer valid.

Return value

Returns true if an element was found.

Exceptions

The given iterator must belong to the collection and must point to an element; otherwise, the exception IteratorInvalid respectively IteratorInBetween is raised.

Inquiring collection information

unsigned long number_of_different_keys ();

Return value

Returns the number of different keys in the collection.

unsigned long number_of_elements_with_key (in any key) raises(KeyInvalid);

Return value

Returns the number elements with key specified.

Exceptions

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The EqualityKeyCollection Interface

interface EqualityKeyCollection : EqualityCollection, KeyCollection{};

Description

This interface combines the interfaces representing the properties “key access” and “element equality.” See “The EqualityCollection Interface” on page 17-37 and “The KeyCollection Interface” on page 17-42.

The KeySortedCollection Interface

```
interface KeySortedCollection : KeyCollection, SortedCollection {
// locating elements
boolean locate_first_element_with_key (in any key, in Iterator where) raises (KeyInvalid,
IteratorInvalid);
boolean locate_last_element_with_key(in any key, in Iterator where) raises (KeyInvalid,
IteratorInvalid);
boolean locate_previous_element_with_key (in any key, in Iterator where) raises (KeyInvalid,
IteratorInvalid);
boolean locate_previous_element_with_different_key(in Iterator where) raises (IteratorInBetween,
IteratorInvalid);
};
```

This interface combines the interfaces representing the properties “key access” and “ordering.” See “The KeyCollection Interface” on page 17-42 and “The SortedCollection Interface” on page 17-37.

Locating elements

boolean locate_first_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);

Description

Locates the first element in iteration order in the collection with the same key as the given key. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_last_element_with_key(in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates the last element in iteration order in the collection with the same key as the given key. Sets the given iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_element_with_key (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between,* locating begins at the iterator's "potential previous" element.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must belong to the collection and be valid; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_element_with_different_key(in Iterator where) raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception KeyInvalid is raised.

The given iterator must point to an element; otherwise, the exception IteratorInBetween or IteratorInvalid is raised.

The EqualitySortedCollection Interface

This interface combines the interfaces representing the properties "element equality" and "ordering." See "The EqualityCollection Interface" on page 17-37 and "The SortedCollection Interface" on page 17-37. It adds those methods which exploit the combination of both properties.

```
interface EqualitySortedCollection : EqualityCollection, SortedCollection {
```

// locating elements

boolean locate_first_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

boolean locate_last_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

boolean locate_previous_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

boolean locate_previous_different_element (in Iterator where) raises (IteratorInvalid);

};

Locating elements

boolean locate_first_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_last_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_element (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between,* the search begins at the iterator's "potential previous" element.

Return value

Returns true if an element was found.

Exceptions

The given element must be of the expected type otherwise the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_different_element (in Iterator where) raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a value different from the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The given iterator must point to an element; otherwise, the exception IteratorInBetween or IteratorInvalid is raised.

The EqualityKeySortedCollection Interface

interface EqualityKeySortedCollection: EqualityCollection, KeyCollection, SortedCollection {};

This interface combines the interface representing the properties "element equality," "key access," and "ordering."

The EqualitySequentialCollection Interface

This interface combines the interface representing the properties "element equality" and "(sequential) ordering" and offers additional operations which exploit this combination.

```
interface EqualitySequentialCollection: EqualityCollection, SequentialCollection
{
// locating elements
boolean locate_first_element_with_value (in any element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);
boolean locate_last_element_with_value (in any element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);
boolean locate_previous_element_with_value (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);
};
```

Locating elements

boolean locate_first_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The given iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_last_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection; otherwise, the exception IteratorInvalid is raised.

boolean locate_previous_element_with_value (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between,* locating begins at the iterators "potential previous" element.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The iterator must belong to the collection and be valid; otherwise, the exception IteratorInvalid is raised.

17.5.4 Concrete Collections Interfaces

The previously listed "abstract views" on collections combine the properties "key access," "element equality," and "ordering relationship" on elements. The subsequent interfaces add "uniqueness" support for "multiples." To reduce the complexity of the hierarchy, this fourth property is not represented by a separate interface.

The KeySet Interface

interface KeySet: KeyCollection {};

The KeySet offers an interface representing the property "key access" with the semantics of "unique keys required." See "The KeyCollection Interface" on page 17-42.

The KeyBag Interface

interface KeyBag: KeyCollection {};

The KeyBag offers the interface representing the property "key access" with multiple keys allowed. See "The KeyCollection Interface" on page 17-42.

The Map Interface

```
interface Map : EqualityKeyCollection {
// set theoretical operations
void difference_with (in Map collector) raises (ElementInvalid);
void add_difference (in Map collector1, in Map collector2)raises (ElementInvalid);
void intersection_with (in Map collector) raises (ElementInvalid);
void add_intersection (in Map collector1, in Map collector2) raises (ElementInvalid);
void union_with (in Map collector) raises (ElementInvalid);
void add_union (in Map collector1, in Map collector2)raises (ElementInvalid);

// testing equality
boolean equal (in Map collector) raises (ElementInvalid);
boolean not_equal (in Map collector) raises(ElementInvalid);
};
```

The Map offers the interface representing the combination of the properties "element equality testable" and "key access" and supports the semantics "unique keys required" (which implies unique elements). See "The EqualityKeyCollection Interface" on page 17-50.

With element equality defined, a test on equality for collections of the same type is possible as well as a meaningful definition of the set theoretical operations.

Set theoretical operations

void difference_with (in Map collector) raises(ElementInvalid);

Description

Makes this collection the difference between this collection and the given collection. The difference of A and B (A minus B) is the set of elements that are contained in A but not in B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the difference of P and Q contains the element X m-n times if "m > n," and zero times if "m <= n."

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Valid iterators pointing to removed elements go *in-between*. All other iterators keep their state.

void add_difference (in Map collector1, in Map collector2) raises (ElementInvalid);

Description

Creates the difference between the two given collections and adds the difference to this collection.

Exceptions

Elements of the given collections must be of the expected type in this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Adding the difference takes place one by one so the semantics for add applies here for raised exceptions and iterator state.

void intersection_with (in Map collector) raises (ElementInvalid);

Description

Makes this collection the intersection of this collection and the given collection. The intersection of A and B is the set of elements that is contained in both A and B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the intersection of P and Q contains the element X "MIN(m,n)" times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Valid iterators of this collection pointing to removed elements go *in-between*.

All other iterators keep their state.

void add_intersection (in Map collector1, in Map collector2) raises (ElementInvalid);

Description

Creates the intersection of the two given collections and adds the intersection to this collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Adding the intersection takes place one by one so the semantics for add apply here for raised exceptions and iterator state.

void union_with (in Map collector) raises (ElementInvalid);

Description

Makes this collection the union of this collection and the given collection. The union of A and B are the elements that are members of A or B or both.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the union of P and Q contains the element X m+n times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Adding takes place one by one so the semantics for add applies here for raised exceptions and iterator state.

void add_union (in Map collector1, in Map collector2) raises (ElementInvalid);

Description

Creates the union of the two given collections and adds the union to the collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

Side effects

Adding the intersection takes place one by one; therefore, the semantics for add applies here for validity of iterators and raised exceptions.

Testing equality

boolean equal (in Map collector) raises(ElementInvalid);

Return value

Returns true if the given collection is equal to the collection.

This operation is defined for other collections, too. Two collections are equal if the number of elements in each collection is the same and if the following conditions (depending on the collection properties) are fulfilled.

- **Collections with unique elements**: If the collections have unique elements, any element that occurs in one collection must occur in the other collections, too.
- **Collections with non-unique elements**: If an element has n occurrences in one collection, it must have exactly n occurrences in the other collection.
- **Sequential collections**: They are sequential collections if they are lexicographically equal based on element equality defined for the elements of the sequential collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception ElementInvalid is raised.

boolean not_equal (in Map collector) raises (ElementInvalid);

Return value

Returns true if the given collection is not equal to this collection.

The Relation Interface

```
interface Relation : EqualityKeyCollection {
// equal, not_equal, and the set-theoretical operations as defined for Map
};
```

The Relation interface offers the interface representing the combination of the properties “element equality testable” and “key access” and supports the semantics “multiple elements allowed.” See “The EqualityKeyCollection Interface” on page 17-50. For a definition of the set-theoretical operation see “The Map Interface” on page 17-57.

The Set Interface

```
interface Set : EqualityCollection {
// equal, not_equal, and the set theoretical operations as defined for Map
};
```

The Set offers the interface representing the property “element equality testable” with the semantics of “unique elements required.” See “The EqualityCollection Interface” on page 17-37.

The Bag Interface

```
interface Bag : EqualityCollection {
// equal, not_equal, and the set theoretical operations as defined for Map
};
```

The Bag offers the interface representing the property “element equality testable” with the semantics of “multiples allowed.” See “The EqualityCollection Interface” on page 17-37.

The KeySortedSet Interface

```
interface KeySortedSet : KeySortedCollection {
long compare (in KeySortedSet collector, in Comparator comparison);
};
```

The KeySortedSet offers the sorted variant of KeySet. See “The KeySortedCollection Interface” on page 17-50.

The sorted variant of KeySet introduces a new operation compare which can be supported only when there is “ordering.” This operation takes an instance of a user-defined Comparator as given parameter. See “The Command and Comparator Interface” on page 17-121.

The Comparator defines the comparison to be used for the elements in the context of this compare operation. Comparison on two KeySortedSets then is a lexicographical comparison based on this element comparison.

long compare (in KeySortedSet collector, in Comparator comparison) raises (ElementInvalid);

Description

Compares this collection with the given collection. Comparison yields:

- <0 if this collection is less than the given collection,
- 0 if the collection is equal to the given collection, and
- >0 if the collection is greater than the given collection.

Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. If such a pair does not exist, the collection with more elements is the greater one.

The "compare" operation of the user's comparator (interface derived from Comparator) must return a result according to the following rules:

>0	if (element1 > element2)
0	if (element1 = element2)
<0	if (element1 < element2)

Return value

Returns the result of the collection comparison.

The KeySortedBag Interface

```
interface KeySortedBag : KeySortedCollection {
long compare (in KeySortedBag collector, in Comparator comparison);
};
```

The KeySortedBag is the sorted variant of the KeyBag. See "The KeySortedCollection Interface" on page 17-50 The additional operation compare is offered. See "The KeySortedSet Interface" on page 17-61.

The SortedMap Interface

```
interface SortedMap : EqualityKeySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedMap collector, in Comparator comparison);
};
```

The SortedMap interface is the sorted variant of a Map. See "The EqualityKeySortedCollection Interface" on page 17-55. The additional operation compare is offered. See "The KeySortedSet Interface" on page 17-61.

The SortedRelation Interface

```
interface SortedRelation : EqualityKeySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedRelation collector, in Comparator comparison);
```

```
};
```

The SortedRelation interface is the sorted variant of a Relation. See “The EqualitySortedCollection Interface” on page 17-52. The additional operation compare is offered. See “The KeySortedSet Interface” on page 17-61.

The SortedSet Interface

```
interface SortedSet : EqualitySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedSet collector, in Comparator comparison);
};
```

The SortedSet interface is the sorted variant of a Set. The additional operation compare is offered. See “The KeySortedSet Interface” on page 17-61.

The SortedBag Interface

```
interface SortedBag: EqualitySortedCollection {
// equal, not_equal, and the set theoretical operations
long compare (in SortedBag collector, in Comparator comparison);
};
```

The SortedBag interface is the sorted variant of a Bag. See “The EqualitySortedCollection Interface” on page 17-52. The additional operation compare is offered. See “The KeySortedSet Interface” on page 17-61.

The Sequence Interface

```
interface Sequence : SequentialCollection {
// Comparison
long compare (in Sequence collector, in Comparator comparison);
};
```

The Sequence supports the interface representing the property “sequential ordering.” This property enables the definition of comparison on two Sequences; therefore, the operation compare is offered. See “The SequentialCollection Interface” on page 17-31.

The EqualitySequence Interface

```
interface EqualitySequence : EqualitySequentialCollection {
// test on equality
boolean equal (in EqualitySequence collector);
boolean not_equal (in EqualitySequence collector);
// comparison
long compare (in EqualitySequence collector, in Comparator comparison);
```

```
};
```

The EqualitySequence supports the combination of the properties “sequential ordering” and “element equality testable.” See “The EqualitySequentialCollection Interface” on page 17-55. This allows the operations equal, not_equal and compare.

The Heap Interface

```
interface Heap : Collection {};
```

The Heap does not support any property at all. It just delivers the basic Collection interface. See “The Collection Interface” on page 17-21.

17.5.5 Restricted Access Collection Interfaces

Common data structures, such as a stack, may restrict access to the elements of a collection. The restricted access collections support these data structures. Stack, Queue, and Dequeue are essentially restricted access Sequences. PriorityQueue is essentially a restricted access KeySortedBag. For convenience, these interfaces offer the commonly used operation names such as push, pop, etc. rather than add_element, remove_element_at. Although the restricted access collections form their own hierarchy, the naming was formed in a way that allows mixing-in with the hierarchy of the combined property collections.

This may be useful to support several views on the same instance of a collection. For example, a “user view” to a job queue with restricted access of a PriorityQueue and an “administrator view” to the same print job queue with the full capabilities of a KeySortedBag.

17.5.6 Abstract RestrictedAccessCollection Interface

The RestrictedAccessCollection Interface

```
// Restricted Access Collections
interface RestrictedAccessCollection {

// getting information on collection state
boolean unfilled ();
unsigned long size ();

// removing elements
void purge ();
};
```

```
boolean unfilled ();
```

Return value

Returns true if the collection is empty.

unsigned long size ();

Return value

Returns the number of elements in the collection.

void purge ();

Description

Removes all elements from the collection. See "The Collection Interface" on page 17-21.

17.5.7 Concrete Restricted Access Collection Interfaces

The Queue Interface

```
interface Queue : RestrictedAccessCollection {

// adding elements
void enqueue (in any element) raises (ElementInvalid);

// removing elements
void dequeue () raises (EmptyCollection);
boolean element_dequeue (out any element) raises (EmptyCollection);
};
```

A Queue may be considered as a restricted access Sequence. Elements are added at the end of the queue only and removed from the beginning of the queue. FIFO behavior is delivered.

Adding elements

void enqueue (in any element) raises (ElementInvalid);

Description

Adds the element as last element to the Queue.

Exceptions

The given element must be the expected type; otherwise, the exception ElementInvalid is raised.

Removing elements

void dequeue () raises (EmptyCollection);

Description

Removes the first element from the queue.

Exceptions

The queue must not be empty; otherwise, the exception EmptyCollection is raised.

boolean element_dequeue(out any element) raises (EmptyCollection);

Description

Retrieves the first element in the queue, returns it via the output parameter element, and removes it from the queue.

Return value

Returns true if an element was retrieved.

Exceptions

The queue must not be empty; otherwise, the exception EmptyCollection is raised.

The Dequeue Interface

```
interface Deque : RestrictedAccessCollection {

// adding elements
void enqueue_as_first (in any element) raises (ElementInvalid);
void enqueue_as_last (in any element) raises(ElementInvalid);

// removing elements
void dequeue_first () raises (EmptyCollection);
boolean element_dequeue_first (out any element) raises (EmptyCollection);
void dequeue_last () raises (EmptyCollection);
boolean element_dequeue_last (out any element) raises (EmptyCollection);
};
```

The Dequeue may be considered as a restricted access Sequence. Adding and removing elements is only allowed at both ends of the double-ended queue. The semantics of the Dequeue operation is comparable to the operations described for the Queue interface. See “The Queue Interface” on page 17-65.

The Stack Interface

```
interface Stack: RestrictedAccessCollection {

// adding elements
void push (in any element) raises (ElementInvalid);

// removing and retrieving elements
void pop () raises (EmptyCollection);
boolean element_pop (out any element) raises (EmptyCollection);
boolean top (out any element) raises (EmptyCollection);
};
```

The Stack may be considered as a restricted access Sequence. Adding and removing elements is only allowed at the end of the queue. LIFO behavior is delivered.

Adding elements

void push (in any element) raises (ElementInvalid);

Description

Adds the element to the stack as the last element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Removing elements

void pop () raises (EmptyCollection);

Description

Removes the last element from the stack.

Exceptions

The stack must not be empty; otherwise, the exception EmptyCollection is raised.

boolean element_pop (out any element) raises (EmptyCollection);

Description

Retrieves the last element from the stack and returns it via the output parameter element and removes it from the stack.

Return value

Returns true if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception EmptyCollection is raised.

Retrieving elements

boolean top (out any element) raises (EmptyCollection);

Description

Retrieves the last element from the stack and returns it via the output parameter element.

Return value

Returns true if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception EmptyCollection is raised.

The PriorityQueue Interface

```
interface PriorityQueue: RestrictedAccessCollection {
// adding elements
void enqueue (in any element) raises (ElementInvalid);

// removing elements
void dequeue () raises (EmptyCollection);
boolean element_dequeue (out any element) raises (EmptyCollection);
};
```

The PriorityQueue may be considered as a restricted access KeySortedBag. The interface is identical to that of an ordinary Queue, with a slightly different semantics for adding elements.

Adding elements

void enqueue (in any element) raises (ElementInvalid);

Description

Adds the element to the priority queue at a position determined by the ordering relation provided for the key type.

Exceptions

The Element must be the expected type; otherwise, the exception ElementInvalid is raised.

Removing elements

void dequeue () raises (EmptyCollection);

Description

Removes the first element from the collection.

Exceptions

The priority queue must be not be empty; otherwise, the exception EmptyCollection is raised.

boolean element_dequeue (out any element) raises (EmptyCollection);

Description

Retrieves the first element in the priority queue and returns it via the output parameter element, removes it from the priority queue, and returns the copy to the user.

Return value

Returns true if an element is retrieved.

Exceptions

The priority queue must not be empty; otherwise, the exception EmptyCollection is raised.

17.5.8 Collection Factory Interfaces

There is one collection factory defined per concrete collection interface which offers a typed operation for the creation of collection instances supporting the respective collection interface as its principal interface.

The information passed to a collection implementation at creation time is:

1. Element type specific information required to implement the correct semantics. For example, to implement Set semantics one has to pass the information how to test the equality of elements.

2. Element type specific information that can be exploited by the specific implementation variants. For example, a hashtable implementation of a Set would exploit the information how the hash value for collected elements is computed.

   This element type specific information is passed to the collection implementation via an instance of a user-defined specialization of the Operations interface.

3. An implementation hint about the expected number of elements collected. An array based implementation may use this hint as an estimate for the initial size of the implementation array.

To enable the support for, and a user-controlled selection of implementation variants, there is a generic extensible factory defined. This allows for registration of implementation variants and their user-defined selection at creation time.

The CollectionFactory and CollectionFactories Interfaces

```
interface Operations;
interface CollectionFactory {
Collection generic_create (in ParameterList parameters) raises (ParameterInvalid);
};
```

CollectionFactory defines a generic collection creation operation which enables extensibility and supports the creation of collection instances with the very basic capabilities.

Collection generic_create (in ParameterList parameters) raises (ParameterInvalid);

Returns a new collection instance which supports the interface Collection and does not offer any type checking. A sequence of name-value pairs is passed to the create operation. The only processed parameter in the given list is "expected_size," of type "unsigned long."

This parameter is optional and gives an estimate of the expected number of elements to be collected.

Note – All collection interface specific factories defined in this specification inherit from the interface CollectionFactory to enable their registration with the extensible generic CollectionFactories factory specified below.

```
interface CollectionFactories : CollectionFactory {
boolean add_factory (in Istring collection_interface, in Istring impl_category, in Istring
impl_interface, in CollectionFactory factory);
boolean remove_factory (in Istring collection_interface, in Istring impl_category, in Istring
impl_interface);
};
```

The interface CollectionFactories specifies a generic extensible collection creation capability. It maintains a registry of collection factories. The create operation of the CollectionFactories does not create collection instances itself, but passes the requests through to an appropriate factory registered with it and passes the result through to the caller. Note that only factories derived from CollectionFactory can be registered with CollectionFactories.

boolean add_factory (in Istring collection_interface, in Istring impl_category, in Istring impl_interface, in CollectionFactory factory);

Registers the factory with three pieces of information:

1. collection_interface specifies the collection interface (directly or indirectly derived from Collection) supported by the given factory. That is, a collection instance created via the given factory has to support the given interface collection_interface.
2. impl_interface specifies the implementation interface (directly or indirectly derived from the interface specified in collection_interface) supported by the registered factory. Collection instances created via this factory are instances of this implementation interface.
3. impl_category specifies a named group of equivalent implementation interfaces to which the implementation interface supported by the registered factory belongs. A group of implementation interfaces of a given collection interface are equivalent if they:
    - rely on the same user-defined implementation support, that is, the same operations defined in the user-defined specialization of the Operations interface.
    - are based on essentially the same data structure and deliver comparable performance characteristics.

The following table lists *examples* of implementation categories (representing common implementations).

Table 17-4 Implementation Category Examples

Implementation Category	**Description**
ArrayBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is an array.
LinkedListBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is a simple linked list.
SkipListsBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a KeyCollection derived from KeyCollection. The basic data structure are skip lists.

HashTableBased	A hash-function has to be defined for key element values that depend on whether or not the interface implemented is a KeyCollection derived from KeyCollection. The basic data structure is a hashtable based on the hash-function defined.
AVLTreeBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a KeyCollection derived from KeyCollection. The basic data structure is an AVL tree.
BStarTreeBased	A compare operation has to be defined for key values. The basic data structure is a B*tree.

The operation does not check the validity of the registration request in the sense that it checks any of the restrictions on the parameters described above, but just registers the given information with the factory. It is the responsibility of the user to ensure that the registration is valid.

The entry is added if there is not already a factory registered with the same three pieces of information; otherwise, the registration is ignored. Returns true if the factory is added.

boolean remove_factory (in Istring collection_interface, in Istring impl_category, in Istring impl_interface)

Description

Removes the factory registered with the given three pieces of information from the registry.

Return value

Returns true if an entry with that name exists and is removed.

create (ParameterList parameters) raises (ParameterInvalid)

The create operation of the CollectionFactories interface does not create instances itself, but passes through creation requests to factories registered with it. The factory is passed a sequence of name-value pairs of which the only mandatory one is collection_interface” of type Istring.

collection_interface” of type Istring	A string which specifies the name of the collection interface (directly or indirectly derived from Collection) the collection instance created has to support. This name-value pair corresponds to the collection_interface parameter of the add_factory() operation.

The following name-value pairs are optional:

“impl_category” of type Istring	A string which denotes the desired implementation category. This name-value pair corresponds to the impl_category parameter of the add_factory() operation.
“impl_interface” of type Istring	A string which specifies a desired implementation interface. This name-value pair corresponds to the impl_interface parameter of the add_factory() operation.

If one or both of these name-value pairs are given, it is searched for a best matching entry in the factory registry and the request is passed through to the respective factory. “Best matching” means that if an implementation interface is given, it is searched for a factory supporting an exact matching implementation interface first. If no factory supporting the desired implementation interface is registered, it is searched for a factory supporting an implementation interface of the same implementation category.

If none of the two name-value pairs are given, the request is passed to a factory registered as default factory for a given “collection_interface.” For each concrete collection interface specified in this specification, there is one collection specific factory defined which serves as default factory and is assumed to be registered with CollectionFactories.

There must be a name-value pair with name “collection_interface” given and a factory must be registered for “collection_interface;” otherwise, the exception ParameterInvalid is raised.

If a desired implementation interface and/or an implementation category is given, a factory with matching characteristics must be registered; otherwise, the exception ParameterInvalid is raised.

For factories specified for each concrete collection interface in this specification, the following additional name-value pairs are relevant:

"operations" of type Operations	An instance of a user-defined specialization of Operations which specifies element- and/or key-type specific operations.
"expected_size" of type unsigned long	is an unsigned long and gives an estimate about the expected number of elements to be collected.

Those parameters are not processed by the create operation of CollectionFactories itself, but just passed through to a registered factory.

The RACollectionFactory and RACollectionFactories Interfaces

```
interface RACollectionFactory {
RestrictedAccessCollection generic_create (in ParameterList parameters) raises (ParameterInvalid);
};
```

The interface RACollectionFactory corresponds to the interface CollectionFactory, but defines an abstract interface.

```
interface RACollectionFactories : RACollectionFactory {
boolean add_factory (in Istring collection_interface, in Istring impl_category, in Istring
impl_interface, in RACollectionFactory factory);
boolean remove_factory (in Istring collection_interface, in Istring impl_category, in Istring
impl_interface);
};
```

The interface RACollectionFactories corresponds to the CollectionFactories interface. It enables the registration and deregistration of collections with restricted access as well as control over the implementation choice for a given restricted access collection at creation time.

The KeySetFactory Interface

```
interface KeySetFactory : CollectionFactory {
KeySet create (in Operations ops, in unsigned long expected_size);
};
```

KeySet create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of KeySet. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-5 Required element and key-type specific user-defined information for KeySetFactory. []- implied by key_compare.

KeySet						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
			x	**[x]**	x	

The KeyBagFactory Interface

interface KeyBagFactory : CollectionFactory {

KeyBag create (in Operations ops, in unsigned long expected_size);

};

KeyBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of KeyBag. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-6 Required element and key-type specific user-defined information for KeyBagFactory. []- implied by key_compare.

KeyBag						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
			x	**[x]**	x	

The MapFactory Interface

interface MapFactory : CollectionFactory {

Map create (in Operations ops, in unsigned long expected_size);

};

Map create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of Map. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-7 Required element and key-type specific user-defined information for MapFactory. []- implied by key_compare.

Map						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
x			**x**	**[x]**	x	

The RelationFactory Interface

interface RelationFactory : CollectionFactory {

Relation create (in Operations ops, in unsigned long expected_size);

};

Relation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of Relation. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-8 Required element and key-type specific user-defined information for RelationFactory.[]- implied by key_compare.

Relation						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
x			**x**	**[x]**	x	

The SetFactory Interface

interface SetFactory : CollectionFactory {

Set create (in Operations ops, in unsigned long expected_size);

};

Set create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of Set. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-9 Required element and key-type specific user-defined information for SetFactory.[]-implied by compare.

Set						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
[x]	x					

The BagFactory Interface

```
interface BagFactory {
Bag create (in Operations ops, in unsigned long expected_size);
};
```

Bag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of Bag. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-10 Required element and key-type specific user-defined information for BagFactory.[]- implied by compare.

Bag						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
[x]	x					

The KeySortedSetFactory Interface

```
interface KeySortedSetFactory {
KeySortedSet create (in Operations ops, in unsigned long expected_size);
};
```

KeySortedSet create (in Operations ops, in unsigned long expected_size)

Creates and returns an instance of KeySortedSet. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-11 Required element and key-type specific user-defined information for KeySortedSetFactory.[]- implied by key_compare.

KeySortedSet						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
			x	**[x]**	**x**	

The KeySortedBagFactory Interface

interface KeySortedBagFactory : CollectionFactory {

KeySortedBag create (in Operations ops, in unsigned long expected_size);

};

KeySortedBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of KeySortedBag. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-12 Required element and key-type specific user-defined information for KeySortedBagFactory.[]- implied by key_compare.

KeySortedBag						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
			x	**[x]**	**x**	

The SortedMapFactory Interface

interface SortedMapFactory : CollectionFactory {

SortedMap create (in Operations ops, in unsigned long expected_size);

};

SortedMap create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of SortedMap. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-13 Required element and key-type specific user-defined information for SortedMapFactory.[]- implied by key_compare.

SortedMap						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
x			**x**	**[x]**	**x**	

The SortedRelationFactory Interface

```
interface SortedRelationFactory : CollectionFactory {
SortedRelation create (in Operations ops, in unsigned long expected_size);
};
```

SortedRelation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of SortedRelation. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-14 Required element and key-type specific user-defined information for SortedRelationFactory.[]- implied by key_compare.

SortedRelation						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
x			**x**	**[x]**	**x**	

The SortedSetFactory Interface

```
interface SortedSetFactory : CollectionFactory {
SortedSet create (in Operations ops, in unsigned long expected_size);
};
```

SortedSet create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of SortedSet. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-15 Required element and key-type specific user-defined information for SortedSetFactory. []- implied by compare.

SortedSet						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
[x]	**x**					

The SortedBagFactory Interface

```
interface SortedBagFactory {
SortedBag create (in Operations ops, in unsigned long expected_size);
};
```

SortedBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of SortedBag. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-16 Required element and key-type specific user-defined information for SortedBagFactory. []- implied by compare.

SortedBag						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
[x]	**x**					

The SequenceFactory Interface

```
interface SequenceFactory : CollectionFactory {
Sequence create (in Operations ops, in unsigned long expected_size);
};
```

Sequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of Sequence. No requirements on the element respectively key operations to be implemented is specified for a Sequence. Nevertheless one still has to pass an instance of Operations as type checking information has to be passed to the collection implementation.

Note – As the Sequence interface represents array as well as linked list implementation of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The EqualitySequence Factory Interface

```
interface EqualitySequenceFactory : CollectionFactory {
EqualitySequence create (in Operations ops, in unsigned long expected_size);
};
```

EqualitySequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of EqualitySequence. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-17 Required element and key-type specific user-defined information for EqualitySequenceFactory.

Equality Sequence						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
x						

Note – As the EqualitySequence interface represents array as well as linked list implementations of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The HeapFactory Interface

```
interface HeapFactory : CollectionFactory {
Heap create (in Operations ops, in unsigned long expected_size);
};
```

Heap create (in Operations ops, in unsigned long expected_size);

Returns an instance of a Heap. No requirements for the element key operations to be implemented is specified for a Heap. Nevertheless, one still has to pass an instance of Operations as type checking information must pass to the collection implementation.

The QueueFactory Interface

```
interface QueueFactory : RACollectionFactory {
Queue create (in Operations ops, in unsigned long expected_size);
};
```

Queue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a Queue. No requirements for the element key operations to be implemented is specified for a Queue. Nevertheless, one still has to pass an instance of Operations as type checking information must pass to the collection implementation.

The StackFactory Interface

```
interface StackFactory : RACollectionFactory {
Stack create (in Operations ops, in unsigned long expected_size);
};
```

Stack create (in Operations ops, in unsigned long expected_size);

Returns an instance of a Stack. No requirements for the element key operations to be implemented is specified for a Stack. Nevertheless, one still has to pass an instance of Operations as type checking information must pass to the collection implementation.

The DequeFactory Interface

```
interface DequeFactory : RACollectionFactory {
Deque create (in Operations ops, in unsigned long expected_size);
};
```

Deque create (in Operations ops, in unsigned long expected_size);

Returns an instance of a Deque. No requirements on the element key operations to be implemented is specified for a Deque. Nevertheless, one still has to pass an instance of Operations as type checking information must pass to the collection implementation.

The PriorityQueueFactory Interface

```
interface PriorityQueueFactory : RACollectionFactory {
PriorityQueue create (in Operations ops, in unsigned long expected_size);
};
```

PriorityQueue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a PriorityQueue. The given instance of Operations passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-18 Required element and key-type specific user-defined information for PriorityQueueFactory. [] - implied by key_compare.

PriorityQueue						
equal	**compare**	**hash**	**key**	**key_equal**	**key_compare**	**key_hash**
			x	**[x]**	**x**	

17.5.9 Iterator Interfaces

Iterators as pointer abstraction

An iterator is in a first approximation of a pointer abstraction. It is a movable pointer to elements of a collection. Iterators are tightly intertwined with collections. An iterator cannot exist independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this and only this collection.

The iterators specified in this specification form an interface hierarchy which parallels the collection interface hierarchy. The supported iterator movements reflect the capabilities of the corresponding collection type.

The top level Iterator interface defines a generic iterator usable for iteration over all types of collections. It can be set to a start position for iteration and moved via a series of forward movements through the collection visiting each element exactly once.

The OrderedIterator is supported by ordered collections only. It "knows about ordering;" therefore, it can be moved in forward and backward direction.

The KeyIterator exploits the capabilities of key collections. It can be moved to an element with a given key value, advanced to the next element with the same key value, or advanced to the next element with a different key value in iteration order.

The KeySortedIterator is created for key collections sorted by key. The iterator can be advanced to the previous element with the same key value or the previous element with a different key value.

The EqualityIterator exploits the capabilities of equality collections. It can be moved to an element with a given value, advanced to the next element with the same element value, or advanced to the next element with a different element value in iteration order.

The EqualitySortedIterator is created for equality collections sorted by element value. The iterator can be advanced to the previous element with the same value or the previous element with a different value.

Iterators and support for generic programming

Iterators go far beyond being simple "pointing devices." There are essentially two reasons to extend the capabilities of iterators.

1. To support the processing of very large collections which allows for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in "finer grains" via an iterator passed to a client.
2. To enrich the iterator with more capabilities strengthens the support for the generic programming model, as introduced with ANSI STL to the C++ world.

You can retrieve, replace, remove, and add elements via an iterator. You can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore an iterator can have a const designation which is set when created. A const iterator can be used for access only.

The reverse iterator semantics is supported. No extra interfaces are specified to support this, but a reverse designation is set at creation time. An ordered iterator for which the reverse designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, *combined* operations and *batch* or *bulk* operations are offered.

Combined operations are combinations of simple iterator operations often used in loops. These combinations support generic algorithms. For example, a typical combination is "test whether range end is reached; if not retrieve_element, advance iterator to next element."

Batch or *bulk* operations support the retrieval, replacement, addition, and removal of many elements within one operation. In these operations, the "many elements" are always passed as a CORBA::sequence of elements.

The Managed Iterator Model

All iterators are managed. The real benefit of being managed is that these iterators never become undefined. Note that "undefined" is different from "invalid." While "invalid" is a testable state and means the iterator points to nothing, "undefined" means you do not know where the iterator points to and cannot inquiry it. Changing the contents of a collection by adding or deleting elements would cause an unmanaged iterator to become "undefined." The iterator may still point to the same element, but it may also point to another element or even "outside" the collection. As you do not know the iterator state and cannot inquiry which state the iterator has, you are forced to newly position the unmanaged iterator, for example, via a set_to_first_element().

This kind of behavior, common in collection class libraries today, seems unacceptable in a distributed multi-user environment. Assume one client removes and adds elements from a collection with side effects on the unmanaged iterators of another client. The other client is not able to test whether there have been side effects on its unmanaged iterators, but would only notice them indirectly when observing strange behavior of the application.

Managed iterators are intimately related to the collection they belong to, and thus, can be informed about the changes taking place within the collection. They are always in a defined state which allows them to be used even though elements have been added or removed from the collection. An iterator may be in the state *invalid*, that is pointing to nothing. Before it can be used it has to be set to a valid position. An iterator in the state *valid* may either point to an element (and be valid for all operations on it) or it may be in the state *in-between*, that is, not pointing to an element but still "remembering" enough state to be valid for most operations on it.

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum within the collection. There is nothing the managed iterator can point to. Nevertheless, managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a set_to_next_element() for example) without resetting it first.

There are some limitations. Once a managed iterator no longer points to an element, it remembers the iteration order in which the element stood before it was deleted. However, it does not remember the element itself. Thus, there are some operations which cannot be performed even though a managed iterator is used.

Consider an iteration over a Bag, for example. If you iterate over all different elements with the iterator operation set_to_next_different_element(), then removing the element the iterator points to leads to an undefined behavior of the collection later on. By removing the element, the iterator becomes *in-between*. The set_to_next_different_element() operation then has no chance to find the next different element as the collection does not know what is different in terms of the current iterator state. Likewise, for a managed iterator in the state *in-between* all operations ending with "..._at" are not defined. The reason is simple: There is no element at the iterator's position - nothing to retrieve, to replace, or to remove in it. This situation is handled by raising an exception IteratorInvalid.

Additionally, all operations that (potentially) destroy the iteration order of a collection invalidate the corresponding managed iterators that have been in the state *in-between* before the operation was invoked. These are the sort() and the reverse() operation.

The Iterator Interface

```
// Iterators


interface Iterator {
```

// moving iterators

boolean set_to_first_element ();

boolean set_to_next_element() raises (IteratorInvalid);

boolean set_to_next_nth_element (in unsigned long n) raises (IteratorInvalid);

// retrieving elements

boolean retrieve_element (out any element) raises (IteratorInvalid, IteratorInBetween);

boolean retrieve_element_set_to_next (out any element, out boolean more) raises (IteratorInvalid, IteratorInBetween);

boolean retrieve_next_n_elements (in unsigned long n, out AnySequence result, out boolean more) raises (IteratorInvalid, IteratorInBetween);

boolean not_equal_retrieve_element_set_to_next (in Iterator test, out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements

void remove_element() raises (IteratorInvalid, IteratorInBetween);

boolean remove_element_set_to_next() raises (IteratorInvalid, IteratorInBetween);

boolean remove_next_n_elements (in unsigned long n, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween);

boolean not_equal_remove_element_set_to_next (in Iterator test) raises (IteratorInvalid, IteratorInBetween);

// replacing elements

void replace_element (in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

boolean replace_element_set_to_next (in any element) raises(IteratorInvalid, IteratorInBetween, ElementInvalid);

boolean replace_next_n_elements (in AnySequence elements, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

boolean not_equal_replace_element_set_to_next (in Iterator test, in any element) raises(IteratorInvalid,IteratorInBetween, ElementInvalid);

// adding elements

boolean add_element_set_iterator (in any element)raises (ElementInvalid);

boolean add_n_elements_set_iterator (in AnySequence elements, out unsigned long actual_number) raises (ElementInvalid);

// setting iterator state

void invalidate ();

// testing iterators

boolean is_valid ();

boolean is_in_between ();

boolean is_for(in Collection collector);

boolean is_const ();

boolean is_equal (in Iterator test) raises (IteratorInvalid);

// cloning, assigning, destroying an iterators

Iterator clone ();

void assign (in Iterator from_where) raises (IteratorInvalid);

void destroy ();

};

Moving iterators

boolean set_to_first_element ();

Description

The iterator is set to the first element in iteration order of the collection it belongs to. If the collection is empty, that is, if no first element exists, the iterator is invalidated.

Return value

Returns true if the collection it belongs to is not empty.

boolean set_to_next_element () raises (IteratorInvalid);

Description

Sets the iterator to the next element in the collection in iteration order or invalidates the iterator if no more elements are to be visited. If the iterator is in the state *in-between,* the iterator is set to its "potential next" element.

Return value

Returns true if there is a next element.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

boolean set_to_next_nth_element (in unsigned long n) raises (IteratorInvalid);

Description

Sets the iterator to the element n movements away in collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between* the movement to the "potential next" element is the first of the n movements.

Return value

Returns true if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

Retrieving elements

boolean retrieve_element (out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed and returns it via the output parameter element.

Return value

Returns true if an element was retrieved.

Exceptions

The iterator must point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the any. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

---

boolean retrieve_element_set_to_next (out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed to and returns it via the output parameter element. The iterator is moved to the next element in iteration order. If there is a next element more is set to true. If there are no more next elements, the iterator is invalidated and more is set to false.

Return value

Returns true if an element was retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

boolean retrieve_next_n_elements (in unsigned long n, out AnySequence result, out boolean more) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves at most the next n elements in iteration order of the iterator's collection and returns them as sequence of anys via the output parameter result. Counting starts with the element the iterator points to. The iterator is moved behind the last element retrieved. If there is an element behind the last element retrieved, more is set to true. If there are no more elements behind the last element retrieved or there are less than n elements for retrieval, the iterator is invalidated and more is set to false. If the value of n is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns true if at least one element is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

boolean not_equal_retrieve_element_set_to_next (in Iterator test, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Compares the given iterator test with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter element, the iterator is moved to the next element, and true is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter element, the iterator is not moved to the next element, and false is returned.

Return value

Returns true if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator test each must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Removing elements

void remove_element () raises (IteratorInvalid, IteratorInBetween);

Description

Removes the element pointed to by this iterator and sets the iterator *in-between*.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have the const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

boolean remove_element_set_to_next() (IteratorInvalid, IteratorInBetween);

Description

Removes the element pointed to by this iterator and moves the iterator to the next element.

Return value

Returns true if a next element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception IteratorInvalid is raised.

The iterator must not have the const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

boolean remove_next_n_elements (in unsigned long n, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween);

Description

Removes at most the next n elements in iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the next element behind the last element removed. If there are no more elements behind the last element removed or there are less than n elements for removal, the iterator is invalidated. If the value of n is 0, all elements in the collection are removed until the end is reached. The output parameter actual_number is set to the actual number of elements removed. If the value of n is 0, all elements in the collection are removed until the end is reached.

Return value

Returns true if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have the const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

boolean not_equal_remove_element_set_to_next(in iterator test) (IteratorInvalid, IteratorInBetween);

Description

Compares this iterator with the given iterator test. If they are not equal the element this iterators points to is removed and the iterator is set to the next element, and true is returned. If they are equal the element pointed to is removed, the iterator is set *in-between*, and false is returned.

Return value

Returns true if this iterator and the given iterator test are not equal when the operations starts.

Exception

This iterator and the given iterator test must be valid otherwise the exception IteratorInvalid or IteratorInBetween is raised.

This iterator and the given iterator test must not have a const designation otherwise the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

Replacing elements

void replace_element (in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

Description

Replaces the element pointed to by the given element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have a const designation; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected element type; otherwise, the ElementInvalid exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception ElementInvalid is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

boolean replace_element_set_to_next(in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the next element. If there are no more elements, the iterator is invalidated.

Return value

Returns true if there is a next element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have a const designation; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected element type; otherwise, the ElementInvalid exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception ElementInvalid is raised.

For positioning properties, see“The Collection Interface” on page 17-21.

boolean replace_next_n_elements(in AnySequence elements, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

Description

Replaces at most as many elements in iteration order as given in elements by the given elements. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter actual_number.

The iterator is moved to the next element behind the last element replaced. If there are no more elements behind the last element replaced or the number of elements in the collection to be replaced is less than the number given elements, the iterator is invalidated.

Return value

Returns true if there is another element behind the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The elements given must be of the expected type; otherwise, the exception ElementInvalid is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception ElementInvalid is raised.

For positioning property see "The Collection Interface" on page 17-21.

boolean not_equal_replace_element_set_to_next (in Iterator test, in any element) raises (IteratorInvalid,IteratorInBetween, ElementInvalid);

Description

Compares this iterator and the given iterator test. If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the next element, and true is returned. If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the next element, and false is returned.

Return value

Returns true if this iterator and the given iterator test are not equal before the operations starts.

Exceptions

This iterator and the given iterator must be valid and point to an element each; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

This iterator must not have a const designation; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected element type; otherwise, the ElementInvalid exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception ElementInvalid is raised.

For positioning property, see "The Collection Interface" on page 17-21.

Adding elements

boolean add_element_set_iterator (in any element) (ElementInvalid);

Description

Adds an element to the collection that this iterator points to and sets the iterator to the added element. The exact semantics depends on the properties of the collection for which this iterator is created.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

Return value

Returns true if the element was added. The element to be added must be of the expected type; otherwise, the exception ElementInvalid is raised.

Exceptions

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception ElementInvalid is raised.

Side effects

All other iterators keep their state.

void add_n_elements_set_iterator (in AnySequence elements, out unsigned long actual_number) (ElementInvalid);

Description

Adds the given elements to the collection that this iterator points to. The elements are added in the order of the input sequence of elements and the delivered semantics is consistent with the semantics of the add_element_set_iterator operation. It is essentially a sequence of add_element_set_iterator operations. The output parameter actual_number is set to the number of elements added.

Setting iterator state

void invalidate ();

Description

Sets the iterator to the state *invalid*, that is, "pointing to nothing." You may also say that the iterator, in some sense, is set to "NULL."

Testing iterators

Whenever there is a precondition for an iterator operation to be checked, there is a test operation provided that enables the user to avoid raising an exception.

boolean is_valid ();

Return value

Returns true if the Iterator is *valid*, that is points to an element of the collection or is in the state *in-between*.

boolean is_for (in Collection collector);

Return value

Returns true if this iterator can operate on the given collection.

boolean is_const ();

Return value

Returns true if this iterator is created with "const" designation.

boolean is_in_between ();

Return value

Returns true if the iterator is in the state *in-between*.

boolean is_equal (in Iterator test) raises (IteratorInvalid);

Return value

Returns true if the given iterator points to the identical element as this iterator.

Exceptions

The given iterator must belong to the same collection as the iterator; otherwise, the exception IteratorInvalid is raised.

Cloning, Assigning, Destroying iterators

Iterator clone();

Description

Creates a copy of this iterator.

void assign (in Iterator from_where) raises (IteratorInvalid)

Description

Assigns the given iterator to this iterator.

Exceptions

The given iterator must be created for the same collection as this iterator; otherwise, the exception IteratorInvalid is raised.

void destroy();

Description

Destroys this iterator.

The OrderedIterator Interface

```
interface OrderedIterator: Iterator {

// moving iterators
boolean set_to_last_element ();
boolean set_to_previous_element() raises (IteratorInvalid);
boolean set_to_nth_previous_element(in unsigned long n) raises (IteratorInvalid);

void set_to_position (in unsigned long position) raises (PositionInvalid);
```

// computing iterator position

unsigned long position () raises (IteratorInvalid);

// retrieving elements

boolean retrieve_element_set_to_previous(out any element, out boolean more) raises (IteratorInvalid, IteratorInBetween);

boolean retrieve_previous_n_elements (in unsigned long n, out AnySequence result, out boolean more) raises (IteratorInvalid, IteratorInBetween);

boolean not_equal_retrieve_element_set_to_previous (in Iterator test, out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements

boolean remove_element_set_to_previous() raises (IteratorInvalid, IteratorInBetween);

boolean remove_previous_n_elements (in unsigned long n, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween);

boolean not_equal_remove_element_set_to_previous(in Iterator test) raises (IteratorInvalid, IteratorInBetween);

// replacing elements

boolean replace_element_set_to_previous(in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

boolean replace_previous_n_elements(in AnySequence elements, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

boolean not_equal_replace_element_set_to_previous (in Iterator test, in any element) raises (IteratorInvalid,IteratorInBetween, ElementInvalid);

// testing iterators

boolean is_first ();

boolean is_last ();

boolean is_for_same (in Iterator test);

boolean is_reverse ();

};

Moving iterators

boolean set_to_last_element();

Description

Sets the iterator to the last element of the collection in iteration order. If the collection is empty (if no last element exists) the given iterator is invalidated.

Return value

Returns true if the collection is not empty.

boolean set_to_previous_element() raises (IteratorInvalid);

Description

Sets the iterator to the previous element in iteration order, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between,* the iterator is set to its “potential previous” element.

Return value

Returns true if a previous element exists.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

boolean set_to_nth_previous_element (in unsigned long n) raises (IteratorInvalid);

Description

Sets the iterator to the element n movements away in reverse collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between*, the movement to the “potential previous” element is the first of the n movements.

Return value

Returns true if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

void set_to_position (in unsigned long position) raises (PositionInvalid);

Description

Sets the iterator to the element at the given position. Position 1 specifies the first element.

Exceptions

Position must be a valid position (i.e., greater than or equal to 1 and less than or equal to number_of_elements()); otherwise, the exception PositionInvalid is raised.

Computing iterator position

unsigned long position () raises (IteratorInvalid, IteratorInBetween);

Description

Determines and returns the current position of the iterator. Position 1 specifies the first element.

Exceptions

The iterator must be pointing to an element of the collection; otherwise, the exception IteratorInvalid respectively IteratorInBetween is raised.

Retrieving elements

boolean retrieve_element_set_to_previous (out any element, out boolean more) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed to and returns it via the output parameter element. The iterator is set to the previous element in iteration order. If there is a previous element, more is set to true. If there are no more previous elements, the iterator is invalidated and more is set to false.

Return value

Returns true if an element was returned.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

boolean retrieve_previous_n_elements(in unsigned long n, out AnySequence result, out boolean more) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves at most the n previous elements in iteration order of this iterator's collection and returns them as sequence of anys via the output parameter result. Counting starts with the element the iterator is pointing to. The iterator is moved to the element before the last element retrieved.

- If there is an element before the last element retrieved, more is set to true.
- If there are no more elements before the last element retrieved or there are less than n elements for retrieval, the iterator is invalidated and more is set to false.
- If the value of n is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns true if at least one element is retrieved.

Exceptions

The iterator must be valid and pointing to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

boolean not_equal_retrieve_element_set_to_previous (in Iterator test, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Compares the given iterator test with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter element, the iterator is moved to the previous element, and true is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter element, the iterator is not moved to the previous element, and false is returned.

Return value

Returns true if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator test each must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

Replacing elements

boolean replace_element_set_to_previous(in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the previous element. If there are no previous elements, the iterator is invalidated.

Return value

Returns true if there is a previous element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have a const designation; otherwise, the exception IteratorInvalid is raised.

The element must be the expected element type; otherwise, the ElementInvalid exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception ElementInvalid is raised.

For positioning properties, see“The Collection Interface” on page 17-21.

boolean replace_previous_n_elements(in AnySequence elements, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

Description

At most, replaces as many elements in reverse iteration order as given in elements. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter actual_number.

The iterator is moved to the element before the last element replaced. If there are no more elements before the last element replaced or the number of elements in the collection to be replaced is less than the number of given elements, the iterator is invalidated.

Return value

Returns true if there is an element before the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The elements given must be of the expected type; otherwise, the exception ElementInvalid is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception ElementInvalid is raised.

For positioning property, see “The Collection Interface” on page 17-21.

boolean not_equal_replace_element_set_to_previous (in Iterator test, in any element) raises (IteratorInvalid,IteratorInBetween, ElementInvalid);

Description

Compares this iterator and the given iterator test.

- If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the previous element, and true is returned.

- If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the previous element, and false is returned.

Return value

Returns true if this iterator and the given iterator test are not equal before the operations starts.

Exceptions

This iterator and the given iterator each must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

This iterator must not have a const designation; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected element type; otherwise, the ElementInvalid exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception ElementInvalid is raised.

For positioning property, see "The Collection Interface" on page 17-21.

Removing elements

boolean remove_element_set_to_previous() raises (IteratorInvalid, IteratorInBetween);

Description

Removes the element pointed to by this iterator and moves the iterator to the previous element.

Return value

Returns true if a previous element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception IteratorInvalid is raised.

The iterator must not have the const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

boolean remove_previous_n_elements (in unsigned long n, out unsigned long actual_number) raises (IteratorInvalid, IteratorInBetween);

Description

Removes at most the previous n elements in reverse iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the element before the last element removed.

- If there are no more elements before the last element removed or there are less than n elements for removal, the iterator is invalidated.
- If the value of n is 0, all elements in the collection are removed until the beginning is reached. The output parameter actual_number is set to the actual number of elements removed.

Return value

Returns true if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The iterator must not have the const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

boolean not_equal_remove_element_set_to_previous(in Iterator test) raises (IteratorInvalid, IteratorInBetween);

Description

Compares this iterator with the given iterator test.

- If they are not equal, the element this iterator points to is removed, the iterator is set to the previous element, and true is returned.
- If they are equal, the element pointed to is removed, the iterator is set *in-between*, and false is returned.

Return value

Returns true if this iterator and the given iterator test are equal when the operation starts.

Exceptions

This iterator and the given iterator test must be valid; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

This iterator and the given iterator test must not have a const designation; otherwise, the exception IteratorInvalid is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

Testing iterators

boolean is_first ();

Return value

Returns true if the iterator points to the first element of the collection it belongs to.

boolean is_last ();

Return value

Returns true if the iterator points to the last element of the collection it belongs to.

boolean is_for_same (in Iterator test);

Return value

Returns true if the given iterator is for the same collection as this.

boolean is_reverse();

Return value

Returns true if the iterator is created with “reverse” designation.

The SequentialIterator Interface

```
interface SequentialIterator : OrderedIterator {
// adding elements
boolean add_element_as_next_set_iterator (in any element) raises(IteratorInvalid, ElementInvalid);
void add_n_elements_as_next_set_iterator(in AnySequence elements) raises(IteratorInvalid,
ElementInvalid);
```

boolean add_element_as_previous_set_iterator(in any element) raises(IteratorInvalid, ElementInvalid);

void add_n_elements_as_previous_set_iterator(in AnySequence elements) raises(IteratorInvalid, ElementInvalid);

};

Adding elements

boolean add_element_as_next_set_iterator (in any element) raises(IteratorInvalid, ElementInvalid);

Description

Adds the element to the collection that this iterator points to (in iteration order) behind the element this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added before the "potential next" element.

Return value

Returns true if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The element added must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All other iterators keep their state.

void add_n_elements_as_next_set_iterator(in AnySequence elements) raises(IteratorInvalid, ElementInvalid);

Description

Adds the given elements to the collection that this iterator points to behind the element the iterator points to. The behavior is the same as n times calling the operation add_element_as_next_set_iterator().

If the iterator is in the state *in-between*, the elements are added before the "potential next" element.

The elements are added in the order given in the input sequence.

boolean add_element_as_previous_set_iterator(in any element) raises(IteratorInvalid, ElementInvalid)

Description

Adds the element to the collection that this iterator points to (in iteration order) before the element that this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added after the "potential previous" element.

Return value

Returns true if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The element added must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All other iterators keep their state.

```
void add_n_elements_as_previous_set_iterator(in AnySequence elements)
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the given elements to the collection that this iterator points to previous to the element the iterator points to. The behavior is the same as n times calling the operation add_element_as_previous_set_to_next().

If the iterator is in the state *in-between*, the elements are added behind the "potential previous" element.

The elements are added in the reverse order given in the input sequence.

The KeyIterator Interface

```
interface KeyIterator : Iterator {
// moving the iterators
boolean set_to_element_with_key (in any key) raises(KeyInvalid);
boolean set_to_next_element_with_key (in any key) raises(IteratorInvalid, KeyInvalid);
boolean set_to_next_element_with_different_key() raises (IteratorInBetween, IteratorInvalid);

// retrieving the keys
boolean retrieve_key (out any key) raises (IteratorInBetween, IteratorInvalid);
boolean retrieve_next_n_keys (out AnySequence keys) raises (IteratorInBetween, IteratorInvalid);
};
```

Moving iterators

boolean set_to_element_with_key (in any key) raises (KeyInvalid);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to the element located or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns true if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean set_to_next_element_with_key (in any key) raises (IteratorInvalid, KeyInvalid);

Description

Locates the next element in iteration order with the same key value as the given key, starting search at the element next to the one pointed to by the iterator. Sets the iterator to the element located.

- If there is no such element, the iterator is invalidated.
- If the iterator is in the state *in-between*, locating starts at the iterator's "potential next" element.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean set_to_next_element_with_different_key () raises (IteratorInBetween, IteratorInvalid)

Description

Locates the next element in iteration order with a key different from the key of the element pointed to by the iterator, starting the search with the element next to the one pointed to by the iterator. Sets the iterator to the located element.

If no such element exists, the iterator is invalidated.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInBetween respectively IteratorInvalid is raised.

Retrieving keys

boolean key (out any key) raises(IteratorInvalid,IteratorInBetween);

Description

Retrieves the key of the element this iterator points to and returns it via the output parameter key.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

boolean retrieve_next_n_keys (in unsigned long n, out AnySequence keys)
raises(IteratorInvalid, IteratorInbetween)

Description

Retrieves the keys of at most the next n elements in iteration order, sets the iterators to the element behind the last element from which a key is retrieved, and returns them via the output parameter keys. Counting starts with the element this iterator points to.

- If there is no element behind the last element from which a key is retrieved or there are less then n elements to retrieve keys from the iterator is invalidated.
- If the value of n is 0, the keys of all elements in the collection are retrieved until the end is reached.

Return value

Returns true if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The EqualityIterator Interface

```
interface EqualityIterator : Iterator {
// moving the iterators
boolean set_to_element_with_value(in any element) raises(ElementInvalid);
boolean set_to_next_element_with_value(in any element) raises(IteratorInvalid, ElementInvalid);
boolean set_to_next_element_with_different_value() raises (IteratorInBetween, IteratorInvalid);
};
```

Moving iterators

boolean set_to_element_with_value (in any element) raises(ElementInvalid);

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns true if an element is found.

Exceptions

The element must be of the expected type; otherwise, the expected ElementInvalid is raised.

boolean set_to_next_element_with_value(in any element) raises (IteratorInvalid, ElementInvalid);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the iterator. Sets the iterator to the located element in the collection.

- If there is no such element, the iterator is invalidated.
- If the iterator is in the state *in-between*, locating is started at the iterator's "potential next" element.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_next_different_element () raises (IteratorInvalid, IteratorInBetween);

Description

Locates the next element in iteration order that is different from the element pointed to. Sets the iterator to the located element, or if no such element exists, the iterator is invalidated.

Return value

Returns true if the next different element was found.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The EqualityKeyIterator Interface

interface EqualityKeyIterator : EqualityIterator, KeyIterator {};

This interface just combines the two interfaces EqualityIterator (see "The EqualityIterator Interface" on page 17-109) and KeyIterator (see "The KeyIterator Interface" on page 17-106).

The SortedIterator Interface

interface SortedIterator : OrderedIterator {};

This interface does not add any new operations but new semantics to the operations.

The KeySortedIterator Interface

// enumeration type for specifying ranges

enum LowerBoundStyle {equal_lo, greater, greater_or_equal};

enum UpperBoundStyle {equal_up, less, less_or_equal};

interface KeySortedIterator : KeyIterator, SortedIterator

{

// moving the iterators

boolean set_to_first_element_with_key (in any key, in LowerBoundStyle style) raises(KeyInvalid);

boolean set_to_last_element_with_key (in any key, in UpperBoundStyle style) raises (KeyInvalid);

boolean set_to_previous_element_with_key (in any key) raises(IteratorInvalid, KeyInvalid);

boolean set_to_previous_element_with_different_key() raises (IteratorInBetween, IteratorInvalid);

// retrieving keys

boolean retrieve_previous_n_keys(out AnySequence keys) raises
(IteratorInBetween, IteratorInvalid);

};

Moving iterators

boolean set_to_first_element_with_key (in any key, in LowerBoundStyle style) raises (KeyInvalid);

Description

Locates the first element in iteration order in the collection with key:

- equal to the given key, if style is equal_lo
- greater or equal to the given key, if style is greater_or_equal
- greater than the given key, if style is greater

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean set_to_last_element_with_key(in any key, in UpperBoundStyle style);

Description

Locates the last element in iteration order in the collection with key:

- equal to the given key, if style is equal_up
- less or equal to the given key, if style is less_or_equal
- less than the given key, if style is less

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean set_to_previous_element_with_key (in any key) raises(IteratorInvalid, KeyInvalid);

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The key must be of the expected type; otherwise, the exception KeyInvalid is raised.

boolean set_to_previous_element_with_different_key() raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInBetween or IteratorInvalid is raised.

Retrieving keys

boolean retrieve_previous_n_keys (in unsigned long n, out AnySequence keys) raises(IteratorInvalid, IteratorInbetween)

Description

Retrieves the keys of at most the previous n elements in iteration order, sets the iterators to the element before the last element from which a key is retrieved, and returns them via the output parameter keys. Counting starts with the element this iterator points to.

- If there is no element previous the one from which the nth key is retrieved or if there are less than n elements to retrieve keys from, the iterator is invalidated.

- If the value of n is 0, the keys of all elements in the collection are retrieved until the beginning is reached.

Return value

Returns true if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInvalid or IteratorInBetween is raised.

The EqualitySortedIterator Interface

interface EqualitySortedIterator : EqualityIterator, SortedIterator {

// moving the iterator

boolean set_to_first_element_with_value (in any element, in LowerBoundStyle style) raises (ElementInvalid);

boolean set_to_last_element_with_value (in any element, in UpperBoundStyle style) raises (ElementInvalid);

boolean set_to_previous_element_with_value (in any elementally) raises (IteratorInvalid, ElementInvalid);

boolean set_to_previous_element_with_different_value() raises (IteratorInBetween, IteratorInvalid);

};

Moving iterators

boolean set_to_first_element_with_value (in any element, in LowerBoundStyle style) raises(ElementInvalid);

Description

Locates the first element in iteration order in the collection with value:

- equal to the given element value, if style is equal_lo
- greater or equal to the given element value, if style is greater_or_equal
- greater than the given element value, if style is greater

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_last_element_with_value(in any element, in UpperBoundStyle style) raises (ElementInvalid);

Description

Locates the last element in iteration order in the collection with value:

- equal to the given element value, if style is equal_up
- less or equal to the given element value, if style is less_or_equal
- less than the given element value, if style is less

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_previous_element_with_value(in any element) raises(IteratorInvalid, ElementInvalid);

Description

Locates the previous element in iteration order with a value equal to the given element value, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_previous_element_with_different_value() raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a value different from the value of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception IteratorInBetween or IteratorInvalid is raised.

The EqualityKeySortedIterator Interface

```
interface EqualityKeySortedIterator: EqualitySortedIterator, KeySortedIterator { };
```

This interface combines the interfaces KeySortedIterator and EqualitySortedIterator. This interface does not add any new operations, but new semantics.

The EqualitySequentialIterator Interface

```
interface EqualitySequentialIterator : EqualityIterator, SequentialIterator
{
// locating elements
boolean set_to_first_element_with_value (in any element) raises (ElementInvalid);
boolean set_to_last_element_with_value (in any element) raises (ElementInvalid);
boolean set_to_previous_element_with_value (in any element) raises (ElementInvalid);
};
```

Moving Iterators

boolean set_to__first_element_with_value (in any element) raises(ElementInvalid);

Description

Sets the iterator to the first element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_last_element (in any element) raises(ElementInvalid);

Description

Sets the iterator to the last element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns true if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

boolean set_to_previous_element_with_value (in any element) raises (IteratorInvalid, ElementInvalid);

Description

Sets the iterator to the previous element in iteration order that is equal to the given element, beginning search at the element previous to the one specified by the iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, search starts at the "potential precious" element.

Return value

Returns true if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception IteratorInvalid is raised.

The element must be of the expected type; otherwise, the exception ElementInvalid is raised.

17.5.10 Function Interfaces

The Operations Interface

```
Interface Operations {

// element type specific information
readonly attribute CORBA::TypeCode element_type;
boolean check_element_type (in any element);
boolean equal (in any element1, in any element2);
long compare (in any element1, in any element2);
```

```
unsigned long hash (in any element, in unsigned long value);

// key retrieval
any key (in any element);

// key type specific information
readonly attribute CORBA::TypeCode key_type;
boolean check_key_type (in any key);
boolean key_equal (in any key1, in any key2);
long key_compare (in any key1, in any key2);
unsigned long key_hash (in any thisKey, in unsigned long value);

// destroying
void destroy();
};
```

The function interface Operations is used to pass a number of other user-defined element type specific information to the collection implementation.

The first kind of element type specific information passed is used for typechecking. There are attributes specifying the element and key type expected in a given collection. In addition to the type information there are two typechecking operations which allow customizing the typechecking in a user-defined manner. The "default semantics" of these operations is a simple check on whether the type code of the given element or key exactly matches the type code specified in the element key type attribute.

Dependent on the properties as represented by a collection interface the respective implementation relies on some element type specific or key type specific information to be passed to it. For example one has to pass the information "element comparison" to implementation of a SortedSet or "key equality" to the implementation of a KeySet to guarantee uniqueness of keys. To pass this information, the Operations interface is used.

The third use of this interface is to pass element or key type specific information relevant for different categories of implementations. (Performing) implementations of associative collections essentially can be partitioned into the categories comparison-based or hashing-based. An AVL-tree implementation for a KeySet (for example) is key-comparison-based; therefore, it relies on key comparison defined and a hash table implementation of KeySet hashing-based (which relies on the information how a hash key values). Passing this information is the third kind of usage of the Operations interface.

The operations defined in the Operations interface are in summary:

- element type checking and key type checking
- element equality and the ordering relationship on elements
- key equality and ordering relationship on keys

- key access
- hash information on elements and keys

In order to pass this information to the collection, a user has to derive and implement an interface from the interface Operations. Which operations you have to implement depends on the collection interface and the implementation category you want to use. An instance of this interface is passed to a collection at creation time and then can be used by the implementation.

Ownership for an Operations instance is passed to the collection at creation time. That is, the same instance of Operations respectively a derived interface cannot be used in another collection instance. The collection is responsible for destroying the Operations instance when the collection is destroyed.

Operations only defines an abstract interface. Specialization and implementation are part of the application development as is the definition and implementation of respective factories and are not listed in this specification.

Element type specific operations

readonly attribute CORBA::TypeCode element_type;

Description

Specifies the type of the element to be collected.

boolean check_element_type (in any element);

Description

A collection implementation may rely on this operation being defined to use it for its type checking. A default implementation may be a simple test whether the type code of the given element exactly matches element_type. For object references, sometimes a check on equality of the type codes is not desired but a check on whether the type of the given element is a specialization of the element_type.

Return value

Returns true if the given element passed the user-defined element type-checking.

boolean equal (in any element1, in any element2);

Return value

Returns true if element1 is equal to element2 with respect to the user-defined semantics of element equality.

Note – If case compare is defined, the equal operation has to be consistently defined (i.e., is implied by the defined element comparison).

long compare (in any element1, in any element2);

Return value

Returns a value less than zero if element1 < element2, zero if the values are equal, and a value greater than zero if element1 > element2 with respect to the user-defined ordering relationship on elements.

unsigned long hash (in any element, in unsigned long value);

Return value

Returns a user-defined hash value for the given element. The given value specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than value.

Note – The definition of the hash function has to be consistent with the defined element equality (i.e., if two elements are equal with respect to the user-defined element equality they have to be hashed to the same hash value).

Computing the key

any key (in any element);

Description

Computes the (user-defined) key of the given element.

Key type specific information

readonly attribute CORBA::TypeCode key_type;

Description

Specifies the type of the key of the elements to be collected.

boolean check_key_type (in any key);

Return value

Returns true if the given key passed the user-defined element type-checking.

boolean key_equal (in any key1, in any key2);

Return value

Returns true if key1 is equal to key2 with respect to the user-defined semantics of key equality.

Note – If case key_compare is defined, the key_equal operation has to be consistently defined (i.e., is implied by the defined key comparison). When both key and element equality are defined, the definitions have to be consistent in the sense that element equality has to imply key equality.

key_compare (in any key1, in any key2);

Return value

Returns a value less than zero if key1 < key2, zero if the values are equal, and a value greater than zero if key1 > key2 with respect to the user-defined ordering relationship on keys.

unsigned long key_hash (in any key, in unsigned long value);

Return value

Returns a user defined hash value for the given key. The given value specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than value.

Note – The definition of the hash function has to be consistent with the defined key equality (i.e., if two elements are equal with respected to the user defined element equality they have to be hashed to the same hash value).

Destroying the Operations instance

void destroy();

Destroys the operations instance.

The Command and Comparator Interface

Command and Comparator are auxiliary interfaces.

A collection service provider may either provide the interfaces only or a default implementation that raises an exception whenever an operation of these interfaces is called. In either case, a user is forced to provide his/her implementation of either the interfaces or a derived interface to make use of them in the operations all_elements_do, and sort.

The Command Interface

An instance of an interface derived from Command is passed to the operation all_elements_do to be applied to all elements of the collection.

```
interface Command {
boolean do_on (in any element);
};
```

The Comparator Interface

An instance of a user defined interface derived from Comparator is passed to the operation sort as sorting criteria.

```
interface Comparator {
long compare (in any element1, in any element2);
};
```

The compare operation of the user's comparator (interface derived from Comparator) must return a result according to the following rules:

>0 if (element1 > element2)

0 if (element1 = element2)

<0 if (element1 < element2)

Appendix A OMG Object Query Service

A.1 Object Query Service Differences

Identification and Justification of Differences

The relationship between the Object Collection Service (OCS) and the Object Query Service (OQS) is two-fold. The Object Query Service uses collections as *query result* and as scope of query evaluation.

The get_result operation of CosQuery::Query for example and the evaluate operation of CosQuery::QueryEvaluator may return a collection as result or may return an iterator to the query result.

There may be a QueryEvaluator implementation that takes a collection instance passed as input parameter to evaluate a query on this collection which specifies the scope of evaluation. The query evaluator implementation relies on the Collection interface and the generic Iterator being supported by the collection passed.

A CosQuery::QueryableCollection is a special case of query evaluator which allows a collection to serve directly as the scope to which a query may be applied. As QueryableCollection is derived from Collection a respective instance can serve to collect a query result to which further query evaluation is applied.

Both usages of collections - as query result and as scope of evaluation - rely on the fact that a minimum collection interface representing a generic aggregation capability is supported as a common root for all collections. Further, they rely on a generic iterator that can be used on collections independent of their type.

Summarizing, Object Query Service essentially depends on a generic collection service matching some minimal requirements. As Object Query Service was defined when there was not yet any Object Collection Service specification available a generic collection service was defined as part of the Query Service specification.

The CosQueryCollection module defines three interfaces:

- CollectionFactory: provides a generic creation capability
- Collection: defines a generic aggregation capability
- Iterator: offers a minimal interface to traverse a collection.

Those interfaces specify the minimal requirements of OQS to a generic collection service. The following discusses whether it is possible to replace CosQueryCollection module by respective interfaces in the CosCollection module as defined in this specification. Differences are identified and justified.

In anticipation of the details given in the next paragraph we can summarize:

- The CosCollection::Collection top level collection interface matches the CosQueryCollection::Collection interface except for minor differences. Collections as defined in the CosCollection module can be used with Query Service.

- The CosCollection::Collection top level collection interface proposes an operation which one may consider as an overlap with the Object Query Service function. The operation all_elements_do which can be considered a special case of query evaluation.
- The CosCollection::Iterator top level iterator interface is consistent with CosQueryCollection::Iterator interface in the sense that operations defined in CosQueryCollecton::Iterator are supported in CosCollection::Iterator. In addition a managed iterator semantics is defined which is reflected in the specified side effects on iterators for modifying collection operations. This differs from the iterator semantics defined in the Object Query Service specification but is considered a requirement in a distributed environment.
- There are a number of operations in the CosCollection::Iterator interface you do not find in the CosQueryCollection::Iterator interface. They are defined in the CosCollection::Iterator interface to provide support for performing distributed processing of very large collections and to support the generic programming model as introduced with ANSI STL to the C++ world.
- The restricted access collections which are part of this proposal do not inherit from the top level CosCollection::Collection interface. They cannot be used with Object Query Service as they are. But this is in the inherent nature of the restricted access semantics of these collections and is not considered to be a problem. Nevertheless, the interfaces of the restricted access collections allow combining them with the collections of the combined property collections hierarchy via multiple inheritance to enable usage of restricted access collections within the Object Query Service. In doing so, the restricted access collections lose the guarantee for restricted access, but only support interfaces offering the commonly used operation names for convenience.
- The CosQueryCollection::CollectionFactory defines the exact same interface as CosCollection::CollectionFactory.

Replacing the interfaces defined in the Object Query Service CosQuery::Collection module by the respective interface defined in this specification, the Object Collection Service enables the following inheritance relationship:

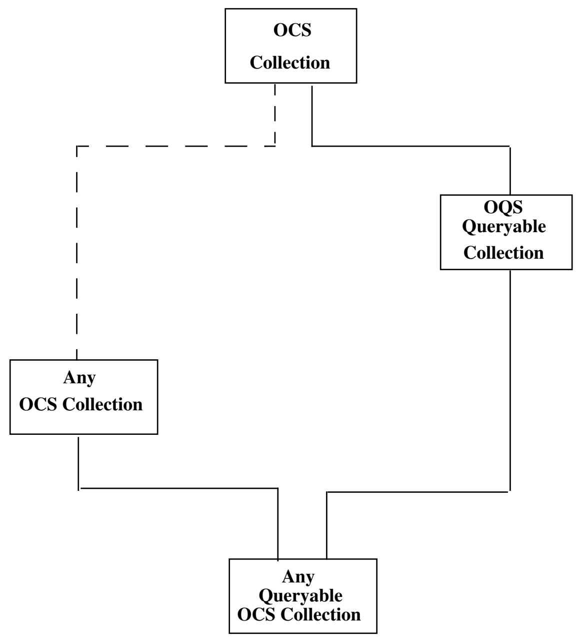


Figure 17-4 Inheritance Relationships

A detailed comparison of the interfaces is given in the following sections and is outlined along the CosQueryCollection module definitions.

CosQueryCollection Module Detailed Comparison

Exception Definitions

The following mapping of exceptions holds true:

- CosQueryCollection::ElementInvalid maps to CosCollection::ElementInvalid
- CosQueryCollection::IteratorInvalid maps to CosCollection::IteratorInvalid (with IteratorInvalidReason not_for_collection)
- CosQueryCollection::PositionInvalid maps to CosCollection::IteratorInvalid (with IteratorInvalidReason is_invalid) and CosCollection::IteratorInBetween

Type Definitions

There are a number of type definitions in the CosQueryCollection module for the mapping of SQL data types and for defining the type Record. These types are Object Query Service specific; therefore, they are not part of the Object Collection Service defined in this specification. Object Query Service may move these definitions to the CosQuery module.

CollectionFactory Interface

The CosQueryCollection::CollectionFactory interface defines the same interface as CosCollection::CollectionFactory and with it the same generic creation capability.

While the generic create operations of CosQueryCollection::CollectionFactory do not raise any exceptions, the respective operation in the CosCollection::CollectionFactory raises exception “ParameterInvalid.”

Collection Interface

The CosQueryCollection::Collection interface defines a basic collection interface, without restricting specializations to any particular type such as equality collections or ordered collections.

Collection Element Type

The element type of Object Query Service collections is a CORBA any to meet the general requirement that collections have to be able to collect elements of arbitrary type. The same holds true for the proposed Object Collection Service defined in this specification.

Using the CORBA any as element type implies the loss of compile time type checking. The Object Collection Service as defined here-in considers support for run-time type checking as important; therefore, it offers respective support. In the interface Collection this is reflected by introducing a read-only attribute “element_type” of type TypeCode which enables a client to inquiry the element type expected.

This differs from Object Query Service collections which do not define any type checking specific support.

Collection Attributes

The following attribute is defined in the OQS Collection interface:

cardinality

This read-only attribute maps to the operation number_of_elements() in CosCollection::Collection. This is semantically equivalent. The name of the operation was chosen consistently with the overall naming scheme of the Collection Service.

Collection Operations

The following operations are defined in the Object Query Service Collection interface.

void add_element (in any element) raises (ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

boolean add_element(in any element) raises (ElementInvalid) *void add_all_elements (in Collection elements) raises (ElementInvalid)*

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void add_all_from (in Collection collector) raises (ElementInvalid).

void insert_element_at (in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid).

void replace_element_at (in any element, in Iterator where) raises (IteratorInvalid, PositionInvalid, ElementInvalid);

This operations maps to

void replace_element_at (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween,ElementInvalid).

void remove_element_at (in Iterator where) raises (IteratorInvalid, PositionInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void remove_element_at (in Iterator where) raises (IteratorInvalid, IteratorInBetween).

void remove_all_elements ()

This operation maps - except for side effects on iterators due to managed iterator semantics - to

unsigned long remove_all ().

any retrieve_element_at (in Iterator where) raises (IteratorInvalid, PositionInvalid)

This operation maps to

boolean retrieve_element_at (in Iterator where, out any element) raises (IteratorInvalid, IteratorInBetween).

Iterator create_iterator ()

This operation maps to

Iterator create_iterator (in boolean read_only).

The parameter "read_only" parameter is used to support const iterators. This is introduced to support the iterator centric ANSI STL like programming model.

Where different operation names are used in the Object Collection Service defined here-in this is done to maintain consistency with the Collection Service overall naming scheme.

Side effects to iterators specified differ from those specified in the Query Service collection module as the Object Collection Service defined here-in specifies a managed iterator model which we consider necessary in a distributed environment. For more details in the managed iterator semantics see chapter "Iterator Interfaces."

The top-level CosCollection::Collection interface proposes all the methods defined in CosQueryCollection::Collection. There are some few additional operations defined in CosCollection::Collection:

boolean is_empty()

This operation is provided as there are collection operations with the precondition that the collection must not be empty. To avoid an exception, the user should have the capability to test whether the collection is empty.

void destroy()

This operation is defined for destroying a collection instance without having to support the complete LifeCycleObject interface.

void all_elements_do(in Command command)

This operation is added for convenience; however, it seems to be an overlap with OQS functionality. This frequently used trivial query should be part of the collection service itself. A typical usage of this operation may be, for example, iterating over the collection to print all element values. Note that the Command functionality is very restricted to enable an efficient implementation. That is, the command is not allowed to change the positioning property of the element applied to and must not remove the element.

Iterator Interface

The CosQueryCollection::Iterator corresponds to CosCollection::Iterator. CosCollection::Iterator is supported for all collection interfaces of the Object Collection Service derived from Collection. The Object Collection Service iterator interfaces defined in this specification are designed to support an iterator centric and generic programming model as introduced with ANSI STL. This implies very powerful iterators which go far beyond simple pointing devices as one needs to be able to retrieve, add, remove elements from/to a collection via an iterator. In addition iterator interfaces are enriched with bulk and combined operations to enable an efficient processing of collections in distributed scenarios. Subsequently, the CosCollection::Iterator is much more powerful than the CosQueryCollection::Iterator.

Iterator Operations

The following operations are defined in the CosQueryCollection::Iterator interface:

- any next () raises (IteratorInvalid, PositionInvalid)

This operation maps to

boolean retrieve_element_set_to_next (out any element) raises (IteratorInvalid, IteratorInBetween)

- void reset ()

This operation maps to

boolean set_to_first_element() of the Object Collection Service Iterator interface.

- boolean more ()

This operation maps to

boolean is_valid() && ! is_inbetween()

Due to the support for iterator centric and generic programming there are number of additional operations in the CosCollection::Iterator interface:

- set_to_next_element, set_to_next_nth_element
- retrieve_element, retrieve_next_n_elements, not_equal_retrieve_element_set_to_next
- remove_element, remove_element_set_to_next, remove_next_n_elements, not_equal_remove_element_set_to_next
- replace_element, replace_element_set_to_next, replace_next_n_elements, not_equal_replace_element_set_to_next
- add_element_set_iterator, add_n_elements_set_iterator
- invalidate
- is_in_between, is_for, is_const, is_equal
- clone, assign, destroy

Most of the operations can be implemented as combinations of other basic iterator operations so that the burden put on Object Query Service providers who implement such an interface should not be too high.

A.2 Other OMG Object Services Defining Collections

There are several object services that define collections, that is Naming Service, Property Service, and the OMG RFC "System Management: Common Management Facility, Volume 1" submission, for example.

These services define very application specific collections. The Naming Service for example defines the interface NamingContext or the Property Service an interface PropertySet. Both are very application specific collections and may be implemented using the Object Collection Service probably wrappering an appropriate Object Collection Service collection rather than specializing one of those collection interfaces.

The collections defined in the System Management RFC form a generic collection service. But the service defines collection members that need to maintain back references to collections in which they are contained to avoid dangling references in collections. This was considered as inappropriate heavyweight for a general object collection service. The collections in the System Management RFC may use Object Collection Service collections for their implementation up to some extent even reuse interfaces.

A.3 OMG Persistent Object Services

Collections as persistent objects in the sense defined by the Persistent Object Service

- may support the CosPersistencePO::PO interface. This interface enables a client being aware of the persistent state to explicitly control the PO's relationship with its persistent data (connect/disconnect/store/restore)
- may support the CosPersistence::SD interface which allows objects to synchronize their transient and persistent data
- have to support one of protocols used to get persistent data in and out of an object, like DA, ODMG, or DDO.

Support for these interfaces does not effect the collection interface.

Persistent *queryable* collections may request index support for collections. "Indexing of collections" enables to exploit underlying indices for efficient query evaluation. We do not consider "indexed collections" as part of the Object Collection Service but think that indexing support can be achieved via composing collections defined in the Object Collection Service proposed.

A.4 OMG Object Concurrency Service

Any implementation of the Object Collection Service probably will have to implement concurrency support. But we did not define any explicit concurrency support in the collection interfaces as part of the Object Collection Service because we consider that as an implementation issue that can be solved by specialization. This also would allow to reuse the respective interfaces of the Object Concurrency Service rather than introducing a collection specific support for concurrency.

Appendix B Relationship to Other Relevant Standards

B.1 ANSI Standard Template Library

The ISO/ANSI C++ standard, as defined by ANSI X3J16 and OSI WG21, contains three sections defining the Containers library, the Iterators library and the Algorithms library, which form the main part of the **S**tandard **T**emplate **L**ibrary. Each section describes in detail the class structure, mandatory methods and performance requirements.

Containers

The standard describes two kinds of container template classes, sequence containers and so called associative containers. There is no inheritance structure relating the container classes.

Sequence containers organize the elements of a collection in a *strictly linear* arrangement. The following sequence containers are defined

- vector: Is a generalization of the concept of an ordinary C++ array the size of which can be dynamically changed. It's an indexed data structure, which allows fast, that is, constant time random access to its elements. Insertion and deletion of an element at the end of a vector can be done in constant time. Insertion and deletion of an element in the middle of the data structure may take linear time.
- deque: Like a vector it is an indexed structure of varying size, allowing fast, that is, constant time random access to its elements. In addition to what a vector offers a deque also offers constant time insertion and deletion of an element at the beginning.
- list: Is a sequence of varying size. Insertion and deletion of an element at any position can be done in constant time. But only linear-time access to an element at an arbitrary position is offered.

Associative containers provide the capability for fast, O(log n), retrieval of elements from the collections by "contents", that is, key value. The following associative containers are provided:

- set: Is a collection of unique elements which supports fast access, O(log n), to elements by element value.
- multiset: Allows multiple occurrences of the same element and supports fast access, O(log n), to elements by value.
- map: Is a collection of (key, value) pairs which supports unique keys.It is an indexed data structure which offers fast, O(log n), access to values by key.
- multimap: Is a collection of (key, value) pairs which allows multiple occurrences of the same key.

Container adapters are the well known containers with restricted access, that is:

- stack

- queue
- priority_queue

As roughly sketched ANSI STL specifies performance requirements for container operations. Those enforce up to some extent the kind of implementation. If you look at the performance requirements for vector, deque and list they correspond to array and list like implementations.

This differs from what the here-in discussed Object Collection Service proposes. The collection classes vector, deque, and list all map to the same interface Sequence. The different performance profiles are delivered via the implementation choice.

Algorithms

Different from other container libraries ANSI STL containers offer a very limited set of operations at the containers themselves. Instead, all higher level operations like union, find, sort, and so on are offered as so called generic algorithms. A generic algorithm is a global template function that operates on all containers - supporting the appropriate type of iterator. There are approximately 50 algorithms offered in ANSI STL.

There are:

- non-mutating sequence algorithms
- mutating sequence algorithms
- sorting and related algorithms
- generalized numeric algorithms

The basic concept here is the separation of data structures and algorithms. Instead of implementing an algorithm for each container in the library you provide a generic one operating on all containers.

If one implements a new container and ensures that an appropriate iterator type is supported one gets the respective algorithms “for free”. One may also implement new generic algorithms working on iterators only which will apply to all containers supporting the iterator type.

In addition, because the algorithms are coded as C++ global template functions, reduction of library and executable size is achieved (selective binding).

Iterators

The key concept in ANSI STL that enables flexibility of STL are Iterator classes. Iterator classes in ANSI STL are C++ pointer abstractions. They allow iteration over the elements of a container.

Their design ensures, that all template algorithms work not only on containers in the library but also on built-in C++ data type array. Algorithms work on iterators rather then on the containers themselves. An algorithms does not even “know” whether it is working with an ordinary C++ pointer or an iterator created for a container of the library.

There are:

- input iterator, output iterator
- forward iterator
- bidirectional iterator
- random access iterator
- const, reverse, insert iterators

Consideration on choice

The collection class concept as defined by the ANSI standard is designed for optimal, local use within programs written in C++. In some sense they are extensions of the language and heavily exploit C++ language features. No considerations, of course, are given to distribution of objects or language neutrality.

Some of the advantages clearly visible in a local C++ environment cannot be carried over into a distributed and language neutral environment. Some of them are even counterproductive.

In summary, the following list of issues are the reason why the ANSI collection class standard has not been considered as a basis for this proposal:

- Aiming with its design at high performance and small code size of C++ applications ANSI STL seems to have avoided inheritance and virtual functions. As no inheritance is defined, polymorphic use of the defined collection classes is not possible.
- The ANSI STL programming model of generic programming is very C++ specific one. ANSI STL containers, iterators, and algorithms are designed as C++ language extension. Containers are smooths extensions of the built-in data type array and iterators are smooth extensions of ordinary C++ pointers. Container in the library are processed by generic algorithms via iterators in the same way as C++ arrays via ordinary pointers. Rather then subclassing and adding operations to a container one extends a container by writing a new generic algorithm. This is a programming model just introduced to the C++ world with ANSI STL and for sure not the programming model Smalltalk programmers are used to.
- As a consequence of the separation of data structures and algorithms containers in ANSI STL up to some extent expose implementation. As an example consider the two sequential containers list and vector. The algorithms sort and merge are methods of the list container. vector on the other hand can support efficient random access and therefore use the generic

algorithms sort and merge. Subsequently you do not find them as methods in the vector interface. This requires rework of clients when server implementations changes from list to vector or deque because of changing access patterns.

- The IDL concept has no notion of global (template) functions. The only conceivable way to organize the algorithms is by collecting them in artificial algorithm object(s). The selective binding advantage is lost in a CORBA environment and careful placement of the algorithm object(s) near the collection must be exercised.
- In the ANSI STL approach the reliance on generic programming as algorithms is substantial. We believe that this concept is not scalable. It is difficult to imagine a generic sort in a CORBA environment is effective without the knowledge of underlying data structures. Each access to a container has to go via an iterator mediated somehow by the underlying request broker, which is not a satisfactory situation.Object Collection Services will be used in an wide variety of environments, ranging from simple telephone lists up to complex large stores using multiple indices, exhibiting persistent behavior and concurrently accessed via Object Query Service. We do not believe that generic algorithms scale up in such environments.

B.1.1 ODMG-93

Release 1.1 of the ODMG specification defines a set of collection templates and an iterator template class.

An abstract base class Collection<T> is defined from which all concrete collections classes are derived. The concrete collection classes supported are Set<T>, Bag<T>, List<T>, Varray<T>. In addition an Iterator class Iterator<T> is defined for iteration over the elements of the collection.

Set and Bag are unordered collections and Bag allows multiples. List is an ordered collection that allows multiples. The Varray<T> is a one dimensional array of varying length.

Collection<T> offers the test empty() and allows to ask for the current number of elements, cardinality(). Further the tests is_ordered() and allows_duplicates() are offered.There is a test on whether an element is contained in a given collection. Operations for insertion, insert_element(), and removal, remove_element() are provided. Last not least there is a remove_all() operation.

Each of the derived classes provides an operator== and an operator!= and an operation create_iterator().

A Set<T> is derived from Collection<T> and offers in addition operations is_subset_off(), is_proper_subset_of(), is_superset_of(), or is proper_superset_of() a suite of set-theoretical operations to form the union, difference, intersection of two sets.

A Bag<T> offers the same interface as Set<T> but allows multiples.

A List<T> offers specific operations to retrieve or remove the first respectively last element in the list or to insert an element as first respectively last element. Retrieving, removing, and replacing an element at a given position is supported. Inserting an element before or after a given position is possible.

Varray<T> exposes the characteristics of a one dimensional array of varying length. An array can be explicitly re-sized. The operator[] is supported. The operations to find, remove, retrieve, and replace an element at a given position are supported.

An instance Iterator<T> is created to iterate over a given collection.The operator= and operator == are defined. There is a reset() operation moving an iterator to the beginning of the collection. There is an operation advance() and overloaded the operator++ to move the iterator to the next element. Retrieving and replacing the element currently "pointed to" is possible. A check on whether iteration is not yet finished is offered, not_done().For convenience in iteration there is an operation next(), combining "check end of iteration, retrieval of an element, and moving to the next element".

ODMG-93 structure is very similar to the proposed Object Collections Service. ODMG-93 Set <T> and Bag<T> correspond very well to Set and Bag as defined herein. List<T> maps one-to-one to an EqualitySequence. A Varray<T> maps to an EqualitySequence too. That the interfaces List<T> and Varray <T> map to the same interface in the Object Collection Service proposed reflects that List<T> and Varray<T> somehow expose the underlying kind of implementation structure assumed - namely a list like structure respectively a table like structure. In the Object Collection Service proposed the different kinds of implementation of a sequence like interface are not reflected in the interface but only in the delivered performance profile. This is the reason why List<T> and Varrary<T> map to the same interface EqualitySequence. The Iterator interface maps to the top level Iterator interface of the iterator hierarchy of the Object Collection Service.

In summary the Object Collection Service proposed is a superset of the ODMG-93 proposed collections and iterators.

Appendix C References

C.1 List of References

OMG, *CORBAservices: Common Object Services Specification*, Volume 1, March 1996.

Index

A
abort
  see rollback
absolute_time 14-9
Abstract Collection Interfaces 17-21
Abstract interface hierarchy 17-4
Abstract Interfaces 16-28
Abstract RestrictedAccessCollection Interface 17-64
accept_security_context 15-164
Access by key 17-3
Access control 15-194
access control model 15-24
Access Decision object 15-118, 15-169
access decision time 15-161
access_allowed 15-118
access_decision 15-111
accountability 15-3
Add Type Operation 16-62
Add_Link Operation 16-51
Additional ObjectID 16-4
Admin Interface 16-70
Administration of Time 14-18
algId 15-257
AlreadyBound 3-9, 3-11
ANSI Standard Template Library 17-131
application components 15-50
application interfaces - level 1 15-289
application interfaces - level 2 15-290
application object xlviii, 4-1
association options 15-280
atomicity 10-49, 10-52, 10-56
  glossary definition 10-85
attribute definitions 15-255
Attribute status 14-16
attribute types 15-254, 15-307
attributes
  control attributes 15-27
  miscellaneous attributes 15-238
  pivilege attributes 15-27
  privilege attributes 15-238
Attributes and Set Operations 16-48
attributeType 15-254
Audit 15-194
audit event families and types 15-309
Audit objects 15-169
audit_channel 15-115
audit_channel_id 15-116
audit_decision 15-112
audit_needed 15-114
audit_write 15-115
auditing application activities 15-68
authenticate 15-91
availability 15-3

B
Bag Interface 17-61
Bag, SortedBag 17-10
BagFactory Interface 17-77
basic key distribution 15-262
basic keys 15-239
basic symmetric key distribution scheme 15-240
Basic Time Service 14-4
basic token format 15-228
bind time 15-161
  client side 15-159
  target side 15-160
binding handle
  securing to the target 15-281
BindingIterator interface 3-12
  next_n operation 3-12
  next_one operation 3-12
bindings 15-52

C
caCertInformation 15-261
callback interface
  described 2-4
call-back object 8-26
cancel_timer 14-16
CannotProceed 3-10
certificate 15-267
check value 15-261
clear_audit_selectors 15-146
CLI 5-34
Collectible elements and the operations interface 17-7
Collectible elements and type safety 17-7
Collectible elements of key collections 17-8
collection 11-4, 11-10
  model 11-12
Collection factories 17-2, 17-5
Collection Factory Interfaces 17-69
Collection Interface 17-21
Collection interface 11-14
  add_all_elements operation 11-17
  add_element operation 11-16
  create_iterator operation 11-18
  insert_element_at operation 11-17
  remove_all_elements 11-18
  remove_element_at operation 11-17
  replace_element_at operation 11-17
  retrieve_element_at operation 11-18
Collection Interface Hierarchies 17-15
Collection interfaces 17-2
CollectionFactory and CollectionFactories Interfaces 17-70
CollectionFactory interface 11-14
Collections 17-2
Combined Collections 17-10
Command and Comparator Interface 17-121
Common collection types 17-2
common contents fields 15-257
common facilities xlviii
common secure interoperability level 0 15-192
common secure interoperability level 1 15-192
common secure interoperability level 2 15-193
compare_time 14-10
CompleteEstablishContext 15-214
component protection 15-56
compound copy request 6-27
compound externalization 2-10, 8-26
compound life cycle 2-9, 9-36, 9-37
  and containment roles 6-42
  and relationship service 6-37, 6-39, 6-41

Index

copy operation example 6-27–6-30
copying, moving relationships 6-39–6-41
copying, moving roles 6-37–6-39
copying, moving, removing nodes 6-35–6-37
copying, moving, removing objects 6-33–6-35
compound name 3-1, 3-2, 3-11, 3-17
compound object 2-2
compound operations 9-36
propagation 9-37
concepts of 2-1
Concrete Restricted Access Collection Interfaces 17-65
concurrency control service
overview 1-3, 7-1
ConcurrencyControl module
OMG IDL 7-8–7-9
confDKuseInfo 15-272
confidentiality 15-3
confKeyDerivationInfo 15-271
confKeySeed 15-271
Conformance Criteria 16-68
Conformance Requirements for Implementation Conformance Classes 16-71
Conformance Requirements for Trading Interfaces as Server 16-69
connect 4-18
Connection interface 5-37
operations 5-37
ConnectionFactory interface 5-37
operations 5-37
Consolidated OMG IDL 14-20, 16-74, 16-93, 16-99
Constraint Language 16-93
Constraint Language BNF 16-95
Constraint Recipe Languag 16-99
consumer 4-2
ConsumerAdmin interface 4-16, 4-17, 4-26
for_consumers operation 4-16
obtain_pull_supplier operation 4-17
obtain_push_supplier operation 4-17
ContainedInRole interface 8-27
containment relationship 9-1, 9-9
defining 9-48–9-49
example 9-23
overview 9-47
ContainsRole interface 8-27
ContextDeleteToken 15-230, 15-252
contextFlags 15-248
continue_authentication 15-92
continue_security_context 15-166
ContinueEstablishContext 15-215
ContinueEstablishToken 15-229
Control interface 10-22
control object 10-21, 10-28, 10-61
Coordinator interface 10-24
create_subtransaction operation 10-28
get_parent_status operation 10-25
get_status operation 10-25
get_top_level_status operation 10-26
get_transaction_name operation 10-28
hash_top_level_tran operation 10-27
hash_transaction operation 10-26
is_ancestor_transaction operation 10-26
is_descendant_transacation operation 10-26
is_related_transaction operation 10-26
is_same_transaction operation 10-26
is_top_level_transaction operation 10-26
register_resource operation 10-27
register_subtran_aware operation 10-27
rollback_only operation 10-28
coordinator object 10-30, 10-33, 10-41, 10-42, 10-53, 10-61
glossary definition 10-86
copy 15-95
CORBA 2-1
documentation set xlix
object references 2-10
standard requests 4-1
CORBA Interfaces 15-200
CORBA OMG IDL based Specification of the Trading Function 16-74
CORBA security facilities and extensibility 15-202
CORBA Standard Security Mechanisms 15-187
CosCompoundExternalization
Node interface 8-7
CosCompoundExternalization module
OMG IDL ??–8-23
CosCompoundLifeCycle module
OMG IDL 6-30–6-33
CosCompoundLifeCycleOperations interface 6-26
CosConcurrencyControl module
overview 7-7
CosContainment module
attributes and operations 9-48–9-49
OMG IDL 9-48
CosEventChannelAdmin module
OMG IDL 4-15–4-16
CosEventComm module
OMG IDL 4-8
CosExternalization module
OMG IDL 8-13–8-14
CosExternalizationContainment module
OMG IDL 8-27
see also CosCompoundExternalization module 8-27
see also CosContainment module 8-27
CosExternalizationReference module
OMG IDL 8-28–8-29
see also CosCompoundExternalization module 8-29
see also CosReference module 8-29
CosGraphs
TraversalCriteria interface 6-41
CosGraphs module 8-26
OMG IDL 9-39–9-41
CosLicensingManager module
OMG IDL for 12-17
CosLifeCycle module
OMG IDL 6-10–6-11
CosLifeCycleContainment module
andCosCompoundLifeCycle and CosContainment modules 6-42
OMG IDL 6-42
CosLifeCycleLifeCycleObject interface 6-37
CosLifeCycleReference module
OMG IDL 6-44
CosNaming module
OMG IDL 3-6–3-8

CosPersistenceDDO module 5-31–5-33
OMG IDL 5-31
CosPersistenceDS_CLI module
OMG IDL 5-35–5-36
CosPersistencePDS module
OMG IDL 5-20
CosPersistencePDS_DA module 5-21–5-29
OMG IDL 5-22
CosPersistencePID module
OMG IDL 5-9
CosPersistencePO module
OMG IDL 5-12
CosPropertyService 13-4
CosQuery module
OMG IDL for 11-23
CosQueryCollection module
OMG IDL for 11-14
CosReference module
attributes and operations 9-50–9-51
CosRelationships module
OMG IDL 9-20–9-23
CosStream module
OMG IDL 8-16–8-20
CosTime 14-4, 14-5
CosTransactions module
datatypes defined by 10-15
OMG IDL 10-69
CosTSInteroperation module
PIDL 10-62, 10-73
CosTypedEventComm module
OMG IDL 4-22
create_invoc_creds_policy 15-113
create_mechanism_policy 15-113
create_qop_policy 15-112
Creating iterators 17-27
creationTime 15-257
credential content at the client 15-197
Credentials 15-60
Credentials object 15-94, 15-169
crypAlgIdentifier 15-260
cryptographic profiles 15-186, 15-226
CSI protocols 15-230
CSI-ECMA hybrid mechanism 15-265
CSI-ECMA protocol 15-237
CSI-ECMA public mechanism 15-269
CSI-ECMA secret key mechanism 15-264
Current 15-60
security operations on 15-105
Current interface 10-41
Cursor interface 5-38
operations 5-38
CursorFactory interface 5-38
operations 5-38
cValues 15-260

D
DA protocol 5-19
compared to ODMG-93 protocol 5-30
DADO 5-26
DAObject interface 5-24
boolean dado_same (inDAObject d) operation 5-24
DataObjectID dado_oid() operation 5-24
PID_DA dado_pid() operation 5-24
void dado_free() operation 5-24
void dado_remove() operation 5-24
DAObjectFactory interface 5-24
DAObjectFactory create() operation 5-25
DAObjectFactoryFinder interface 5-25
find_factory operation 5-25
Data Definition Language
see DDL
data objects 5-27, 5-28
and dynamic access to attributes 5-28
data structures 15-253
Data Types
for Non-repudiation Policy Management Interfaces 15-154
data types 15-90
datastore 5-7, 5-13, 5-17, 5-18, 5-26, 5-34, 5-43
and DDO protocol 5-31
Datastore_CLI interface 5-40
and CLI 5-43
operations 5-41–5-43
DCE association options reduction algorithm 15-280
DCE RPC authorization services 15-278
DCE RPC protection levels 15-279
DCE RPC Security Services 15-278
DCE security parameters 15-280
DCE-CIOP
IOR Security Components for 15-274
DDL 5-21, 5-26, 5-27, 5-28
DDO
storing,restoring,deleting 5-40
DDO interface
attributes 5-32
short add_data() operation 5-32
short add_data_property (in short data_id) operation 5-32
short get_data_count() operation 5-32
short get_data_property_count (in short data_id) operation 5-33
void get_data operation 5-33
void get_data_property operation 5-33
void set_data operation 5-33
void set_data_property operation 5-33
DDO protocol 5-19, 5-30
define 13-9, 13-10, 13-16
Defining 13-9, 13-15
defining and modifying properties 13-9
definingAuthority 15-254
delegation 15-30
privilege delegation 15-31
delegation options
specifying 15-35
technology support for 15-35
delegation schemes
overview of 15-31
delete 9-30, 13-12
Deleting 13-12
deleting properties 13-12
Deque 17-14
DequeFactory Interface 17-82
Dequeue Interface 17-66
Describe Link Operation 16-52
Describe Operation 16-41

Index

Describe Proxy Operation 16-58
Describe Type Operation 16-65
design goals, of event service interfaces 1-2
destroy 3-18, 15-95
destroy operation 3-12
Destroying 13-20
Destroying a collection 17-27
destroying the iterator 13-19, 13-20
Determining 13-13
determining defined property 13-13
dialogue key block 15-271
dialogue keys 15-239
dialogueKeyBlock 15-249
direct access protocol
  see PDS_DA protocol
direct attribute protocol
  see DA protocol
directionIndicator 15-251
DiscardContext 15-215
distributed object system
  threats in 15-3
distributed objects 6-3
Domain Access Policy object 15-137
domain objects 15-53
domains
  and interoperability 15-42
  at object creation 15-76, 15-77
  environment domains 15-56
  federated policy domains 15-39
  ORB Technology Domains
    interoperating between 15-43
  overlapping policy domains 15-40
  policy domain hierarchies 15-39
  policy domains 15-131
  security administration domains 15-228
  security environment domains 15-41
    managing 15-44
  security policy domains 15-38
    interoperating between 15-176
    managing 15-44
  security technology domains 15-42
    interoperating between 15-43
    managing 15-45
dynamic data object protocol
  see DDO protocol
Dynamic Property Evaluation interface 16-67
Dynamic Property Module 16-88
DynamicAttributeAccess interface 5-28
  any attribute_get(in string name) operation 5-28
  AttributeNames attribute_names() operation 5-28
  void attribute_set(in string name, in any value) operation 5-28

E
edge structure 9-46
EdgeIterator interface 9-46
  destroy operation 9-47
  next_n operation 9-47
  next_one operation 9-47
encryptedPlainKey 15-267
Enum ComparisonType 14-7
Enum EventStatus 14-14
Enum OverlapType 14-7
Enum TimeComparison 14-7
Enum TimeType 14-14
Equality collection 17-3
EqualityCollection Interface 17-37
EqualityIterator Interface 17-109
EqualityKeyCollection Interface 17-50
EqualityKeyIterator Interface 17-110
EqualityKeySortedCollection Interface 17-55
EqualityKeySortedIterator Interface 17-115
EqualitySequence 17-11
EqualitySequence Factory Interface 17-81
EqualitySequence Interface 17-63
EqualitySequentialCollection Interface 17-55
EqualitySequentialIterator Interface 17-115
EqualitySortedCollection Interface 17-52
EqualitySortedIterator Interface 17-113
ErrorToken 15-229, 15-246, 15-250
EstablishContext 15-214
establishing credentials 15-59
etContents 15-250
event channel 1-2, 2-2, 2-3, 4-5, 4-13
  adding consumers 4-16
  adding consumers to 4-17
  adding consumers to typed 4-26
  adding pull consumer to typed 4-28
  adding pull consumers to 4-18
  adding pull suppliers to 4-18
  adding push consumers to 4-19
  adding push suppliers to 4-17
  adding push suppliers to typed 4-27
  adding suppliers 4-16
  adding suppliers to 4-17
  adding suppliers to typed 4-27
  and CORBA requests 4-10
  decoders 4-31
  defined 4-2, 4-10
  encoders 4-31
  filtering 4-28–4-29
  implementing typed 4-30–4-31
  sample use 4-32–4-33
event communication
  mixed 4-11
  multiple 4-12
  pull model 1-2, 4-2, 4-7, 4-11
  push model 1-2, 4-2, 4-6, 4-10
  typed pull model 4-20
  typed push model 4-19
event consumer 4-2, 4-6, 4-10
  proxy 4-13
event service
  and CORBA scoping 4-5
  and license service 12-13, 12-15
  design goal of interfaces 1-2
  overview 1-2, 4-1
event supplier 4-2, 4-6, 4-10
  proxy 4-13
event_time 14-17
EventChannel interface 2-2, 4-13, 4-16
exception 4-27
Exceptions 16-23

Additional Exceptions for Link Interface 16-26
Additional Exceptions for Lookup Interface 16-24
Additional Exceptions for Proxy Offer Interface 16-27
Additional Exceptions For Register Interface 16-25
For CosTrading module 16-23
exceptions
described 2-4
InvalidName 3-10
Exceptions and Type Definitions 17-19
export 16-2
Export Operation 16-39
Export Proxy Operation 16-55
Exporter 16-4
Exporter Policies 16-18
Extended Time Service 14-26
external control values construct 15-260
external security services
linking to 15-171
externalization
defined 8-2
externalization service
and compound life cycle 8-7
and inheritance and use of objects 8-8
and life cycle service 2-10
and persistent object service 8-20
and relationship service 2-10, 8-26
and transaction service 8-20
interface summary 8-11
overview 1-4
externalizing a node 8-23
externalizing a relationship 8-25
externalizing a role 8-24

F
facilities
description of 15-90
factory finder 6-7, 6-13, 6-21, 8-3
factory keys
and kind field 6-14, 6-16
factory object 1-2, 6-4
definition 6-18
FactoryFinder interface 6-8, 6-13–6-14
find_factories operation 6-13
feature packages 15-12
and modules 15-17
Federated query example 16-19
FileStreamFactory interface 8-9, 8-13, 8-14
create operation 8-14
flags 15-249
form_complete_evidence 15-130
framework 11-10
Friendly Time Object 14-26
Full-service Trader 16-73
Fully Describe Type Operation 16-65
Function Interfaces 17-3, 17-117
Functional Interfaces 16-30

G
general security data module 15-283
generate_token 15-125
generic factory
criteria parameters 6-17–6-18
generic factory interface 6-5
GenericFactory interface 6-14–6-18, 6-22
and criteria parameter 6-17
and criteria parameters 6-17
create_object operation 6-15, 6-17
supports operation 6-16
get 13-11, 13-12, 13-14, 13-15, 13-17, 13-18
get_all_properties 13-12
get_all_property_names 13-11
get_association_options 15-151
get_attributes 15-98, 15-107
get_audit_selectors 15-147
get_component operation 3-16
get_credentials 15-109
get_delegation_mode 15-152
get_domain_managers 15-100, 15-103
get_effective_rights 15-136
get_NR_features 15-124
get_NR_policy_info 15-154
get_number_of_properties 13-11
get_policy 15-100, 15-103, 15-110
get_properties 13-11
get_property_value 13-11
get_required_rights 15-135
get_rights 15-143
get_security_features 15-96
get_security_names 15-112
get_supported_mechs 15-165
get_token_details 15-128
Getting 13-17
global identifier 2-4
goals of secure DCE-CIOP 15-273
grant_rights 15-141
graphical notation 2-3
graphs of related objects
copying to 6-33
creating traversal criteria for 8-26
destroying 6-35
examples 9-33
moving 6-34
removing 6-34
traversal of 9-35, 9-37
traversing 9-36
GSS Kerberos protocol 15-234
GSS-API 15-203
gss-key-estb-alg 15-268

H
hashAlgId 15-257, 15-261, 15-262
hashedName 15-267
Heap 17-11
Heap Interface 17-64
HeapFactory Interface 17-81
hniIssuingKDS 15-267
hybrid iInter-domain key distribution scheme data elements 15-266
Hybrid Interdomain Scheme
Profile of Ticket as Used in 15-268

Index

I
ictContents 15-247
ictSeal 15-247
IDAPI standard 5-34
import 16-2
ImportAttributes 16-29
Importer 16-4
Importer Policies 16-17
init_security_context 15-162
initial context token 15-233, 15-236, 15-246
InitialContextToken 15-246
initiatorAddress 15-248
inner context tokens 15-229
integDKuseInfo 15-271
integKeyDerivationInfo 15-271
integKeySeed 15-271
integrity 15-3
interceptors
  Access Control Interceptor 15-156
  access control interceptor 15-161
  Secure Invocation Interceptor 15-156
  secure invocation interceptor 15-158
Interface Hierarchies 17-15
interface inheritance.see subtyping
interface repository 2-7
Interfaces
  SecurityLevel2
    AuditDecision Interface 15-114
interfaces
  application developer's interface 15-88
  for non-repudiation 15-296
  implementation-Level Security Object interfaces 15-161
  Implementor's Security interfaces 15-156
  NRservice
    NRCredentials interface 15-124
    NRPolicy interface 15-154
  security administration interface 15-294
  security replaceable service interfaces 15-300
  SecurityAdmin
    AccessPolicy interface 15-136
    AuditPolicy interface 15-144
    DelegationPolicy interface 15-152
    DomainAccessPolicy interface 15-141
    SecureInvocationPolicy interface 15-150
  SecurityLevel1
    Current interface 15-107
  SecurityLevel2
    AccessDecision interface 15-118
    AuditChannel interface 15-115
    Credentials interface 15-95
    Current interface 15-108
    PrincipalAuthenticator Interface 15-91
    RequiredRights interface 15-134
  SecurityReplaceable
    SecurityContext interface 15-166
    Vault interface 15-162
intermediate objects 15-64
internalization
  object's model 8-6
internalizing a node 8-23
internalizing a relationship 8-25
internalizing a role 8-24
interoperability model 15-173
interval 14-10
InvalidName exception 3-10
Invocation Credentials 15-100
invocation credentials 15-105
Invocation Credentials Policy 15-102
invocation policy objects
  client-side 15-101
invocation time policies 15-158
IOR encoding 15-235, 15-244
is_valid 15-98, 15-168
issuerCAName 15-261
issuerDomain 15-257
issuerIdentity 15-257
Iterating over a collection 17-26
Iterator Hierarchy 17-18
Iterator interface 11-14
  any next operation 11-18
  boolean more operation 11-19
  void reset operation 11-18
Iterator Interfaces 17-3, 17-83
Iterators 17-5
Iterators and performance 17-6, 17-84
Iterators and support for generic programming 17-84
Iterators as pointer abstraction 17-83

K
kdSchemeOID 15-263
Key collection 17-3
Key collections 17-8
Key Distribution Schemes 15-263
key distribution schemes 15-239
key distribution types 15-184
key establishment algorithm 15-268
key establishment algorithms 15-227
key establishment data elements 15-267
KeyBag Interface 17-57
KeyBag, KeySortedBag 17-11
KeyBagFactory Interface 17-75
keyId 15-262
KeyIterator Interface 17-106
KeySet Interface 17-57
KeySet, KeySortedSet 17-12
KeySetFactory Interface 17-74
keySize 15-271
KeySortedBag Interface 17-62
KeySortedBagFactory Interface 17-78
KeySortedCollection Interface 17-50
KeySortedIterator Interface 17-110
KeySortedSet Interface 17-61
KeySortedSetFactory Interface 17-77
krb5Ticket 15-266
kying iInformation syntax 15-262

L
Legal Property Value Types 16-94
library names
  PIDL operations 3-18
license service
  and event service 12-13, 12-15

and life cycle service 12-19
and properties service 12-24
and relationship service 12-27
and security service 12-27
example implementation 12-28
exceptions 12-19
overview 12-8
sample implementation 12-14
LicenseServiceManager interface 12-13, 12-17
check_use operation 12-13
end_use operation 12-13
obtain_producer_specific_license_service operation 12-19, 12-28
start_use operation 12-13
licensing attributes
examples of 12-25
life cycle service
and license service 12-19
and naming service 2-9, 6-15
and relationship service 2-9
client's model 6-4
overview 1-2, 6-1, 6-21
LifeCycleObject interface 1-2, 6-6, 6-11–6-13, 6-22, 6-25
and crieteria parameter 6-17
copy operation 6-11
move operation 6-12
NoFactory exception for copy operation 6-11
remove operation 6-13
Link 16-49
Link Creation Policies 16-18
Link Interface 16-70
Link Traversal Control 16-18
LinkAttributes 16-30
Linked Trader 16-72
Linking Traders 16-3
Links 16-11
List Offers Operation 16-48
List Proxies Operation 16-48
List Types Operation 16-64
Listing 13-10
listing and getting properties 13-10
LName interface 3-3, 3-15
delete_component operation 3-17
destroy operation 3-16
equal operation 3-17
insert_component operation 3-16
less_than operation 3-17
num_components operation 3-17
LNameComponent interface 3-3, 3-13, 3-15
get_id operation 3-15
get_kind attribute 3-3
get_kind operation 3-15
set_id operation 3-15
set_kind operation 3-15
LockCoordinator interface 7-9
drop_locks operation 7-10
locks 1-4, 2-7, 7-1, 7-2–7-7
and nested transactions 7-6
intention read and write 7-4
mode compatibility 7-5
multiple possession semantics 7-5
read,write,upgrade 7-4
transaction-duration 7-6
LockSet interface 7-9, 7-10–7-11
change_model operation 7-11
get_coordinator operation 7-11
lock operation 7-11
try_lock 7-11
unlock operation 7-11
LockSetFactory interface 7-13
create operation 7-13
create_related operation 7-13
create_transactional operation 7-13
create_transactional_related operation 7-13
Lookup 16-30
Lookup Interface 16-69

M
Managed Iterator Model 17-84
Managed iterators 17-6
Map Interface 17-57
Map, SortedMap 17-12
MapFactory Interface 17-75
Mapping
to Local Attribute Values 15-200
mapping of common elements to the SECIOP protocol 15-228
mapping security names to externally valid identities 15-199
Mask Type Operation 16-66
MD5 message digest algorithm 12-31
mechanism identifiers 15-244
Mechanism Policy 15-101
mechanism tags 15-225
mechanism types 15-201
message definitions 15-214
Message Protection 15-160
message protection 15-22
MessageError 15-216
MessageInContext 15-216
messages
protecting messages 15-175
methodId 15-259
methodParams 15-260
MICToken 15-230, 15-246, 15-252
miscellaneousAtts 15-259
model
implementing 15-45
model for use and contents of credentials 15-196
models
administrative model 15-75
security object models 15-58
Modify Link Operation 16-53
Modify Operation 16-42

N
name 3-2
binding 3-1
binding operations 3-8
component attributes 3-2
components 3-2
compound 3-2
resolution 3-1
simple 3-2

Index

structure 3-18
name binding 3-1
name component
attributes 3-15
nameHashingAlg 15-267
names library 1-1, 3-3, 3-13
PIDL 3-13–3-14
namespace adminstration 3-5
name-to-object association 3-1
naming context 1-1, 3-1, 3-5, 3-6
and property lists 2-5
deleting 3-11
naming graph 3-1
example 3-2
naming service
and internationalization 3-3, 3-6
design of 3-4
overview 1-1
NamingContext interface 3-8, 3-13, 3-18
bind operation 3-8
bind_context operation 3-9
bind_new_context operation 3-11
destroy operation 3-11
list operation 3-12
new_context operation 3-11
rebind operation 3-8
rebind_context operation 3-9
resolve operation 3-9
unbind operation 3-10
nested queries 11-20
nested transaction 2-10
new_interval 14-12
new_universal_time 14-12
next 13-19, 13-20
Node interface 6-35, 9-35, 9-44
add_role operation 9-45
copy operation 6-35
externalize_node operation 8-23
internalize_node operation 8-23
move operation 6-36
related_object attribute 9-45
remove operation 6-37
remove_role operation 9-45
roles_of_node attribute 9-45
roles_of_type operation 9-45
NodeFactory interface 9-46
create_node operation 9-46
nodes
creating 9-46
NoFactory 6-40
non-repudiation 15-35, 15-70, 15-170
non-repudiation credentials 15-106
non-repudiation service data types 15-123
NotCopyable 6-40
NotMovable 6-40
NotRemovable 6-37

O
Object Management Group xlvii
address of xlix
object model xlix
object references 15-52
object request broker xlviii
object service
context xlviii
specification defined xlix
ODBC standard 5-34
ODMG-93 17-134
ODMG-93 protocol 5-19, 5-30, 5-43, 10-83
integration with transaction service 10-84
Offer Id Iterator 16-45
Offer Identifier 16-9
Offer Iterator 16-35
Offer Selection 16-9
OMG 13-3
OMG Constraint Language BNF 16-93
OMG Constraint Recipe Language 16-99
OMG IDL xlix, 2-2, 3-3
OMG Trading Function Module 16-74
OperationFactory interface
create_compound_operations operation 6-33
operations 3-15
Operations Interface 17-7, 17-117
Operations interface 6-33
copy operation 6-33
destroy operation 6-35
move operation 6-34
remove operation 6-34
OperationsFactory interface 6-33
Operator Restrictions 16-94
OQL-93 Basic Query Language 11-7
OQL-93 Query Language 11-6
ORB security implementors
security replaceability for 15-203
ORB security services 15-80
ORB services 15-51
Ordering of elements 17-3
OSI TP protocol 10-80
exported transactions 10-82
imported transactions 10-81
transaction identifiers 10-81
overlaps 14-11
owfId 15-271
own credentials 15-106
own_credentials 15-110

P
PAC format 15-256
PAC Protection and Delegation 15-242
pacAndCVs 15-248
PACs
specific certificate contents for 15-257
pacSyntaxVersion 15-258
pacType 15-258
PDS 5-43
see persistent data service
PDS interface 5-19–5-20
and DA protocol 5-25
PDS connect operation 5-20
void delete operation 5-20
void disconnect operation 5-20
void restore operation 5-20

void store operation 5-20
PDS_ClusteredDA interface 5-29
ClusterID cluster_id() operation 5-29
ClusterIDs clusters_of() operation 5-29
PDS_ClusteredDA copy_cluster(in PDS_DA source) operation 5-29
PDS_ClusteredDA create_cluster(in string kind) operation 5-29
PDS_ClusteredDA open_cluster(in ClusterID cluster) operation 5-29
string cluster_kind() operation 5-29
PDS_DA interface 5-21, 5-25
and ODMG-93 protocol 5-30
DAObject get_data() operation 5-25
DAObject lookup(in DAObjectID id) operation 5-25
DAObjectFactoryFinder data_factories() operation 5-26
PID_DA get_object_pid(in DAObject dao) operation 5-25
PID_DA get_pid() operation 5-25
void set_data(in DAObject new_data) operation 5-25
PDS_DA protocol 5-21, 5-25
and data objects 5-26
per-message tokens 15-250
persistent data service 5-7, 5-17, 5-26, 5-27
overview 5-18
persistent data service interface
see PDS interface
persistent identifier 5-7
compared to CORBA object reference 5-9
persistent object interface
see PO interface
persistent object manager 5-11
and PO interface 5-13
purpose of 5-17
persistent object service
and clients 5-5
and CORBA accessor operations 5-27
and CORBA Dynamic Invocation interface 5-28
and CORBA persistent reference handling 5-2, 5-3
and datastore 5-6
and factory finders 5-25
and factory objects 5-24
and object implementation 5-6
and persistent data service 5-6
and query service 5-42
and transaction service 5-42
overview 1-3
PID
see persistent identifier
PID interface 5-8
PID_CLI interface 5-38
attributes 5-39
PID_DA interface 5-23
DAObjectID attribute 5-23
PIDL 2-13, 3-3
plainKey 15-267
pmtSeal 15-251
PO interface 5-12–5-13
... connect operation 5-13
void delete operation 5-13
void disconnect operation 5-13
void restore operation 5-13
void store operation 5-13
policies
access policies 15-26, 15-133, 15-189
access policy 15-82
application access policies 15-67
application access policy 15-26
application-enforced policies 15-40
audit policies 15-28, 15-189
object invocation access policy 15-25
Secure Invocation Policies 15-148
Security Policies 15-131
security policies 15-70, 15-75
security policies and domain objects 15-53
security policy 15-80
system-enforced policies 15-40
policiesaudit policies 15-144
POM interface
...connect operation 5-16
OMG IDL 5-16
void delete operation 5-16
void disconnect operation 5-16
void restore operation 5-16
void store operation 5-16
PPID method 15-243
ppidType 15-265
ppidValue 15-265
Preferences 16-10
principal authentication 15-170
Principal Authenticator 15-60
principal_authenticator 15-111
principals
and their security attributes 15-20
PriorityQueue 17-14
PriorityQueue Interface 17-68
PriorityQueueFactory Interface 17-82
privileges 15-258
ProducerSpecificLicenseService interface 12-13, 12-14, 12-17
check_use operation 12-20, 12-21, 12-28
end_use operation 12-20, 12-28
start_use operation 12-20, 12-28
propagation 10-34–10-38, 10-41, 10-44, 10-60, 10-63, 10-65, 10-66
deep 9-37
glossary definition 10-88
none 9-38
shallow 9-37
propagation context 2-13
PropagationCriteriaFactory interface 8-26
create operation 6-41, 8-26
Properties 16-7
Dynamic 16-8
modifiable 16-8
properties
defining and modifying with modes 13-15
properties service
and license service 12-24
PropertiesIterator 13-18
PropertiesIterator interface 13-18
Property 13-23
property list 4-1, 12-24
property modes
getting and setting 13-17

Index

property service
  object classification 13-1
  object usage count 13-1
Property service IDL 13-23
PropertyNamesIterator 13-19
PropertyNamesIterator interface 13-19
PropertySet 13-9
PropertySetDef 13-14
PropertySetDef interface 13-14
PropertySetDefFactory 13-21
PropertySetDefFactory interface 13-21
PropertySetFactory 13-20
PropertySetFactory interface 13-20
protect_message 15-167
protection methods 15-259
protectionMethods 15-258
protocol initialization 15-209
protocol state 15-207
proxies and Time 14-23
Proxy 16-54
Proxy Interface 16-70
Proxy Trader 16-73
ProxyPullConsumer interface 4-18
  connect_pull_supplier operation 4-18
ProxyPullSupplier 4-18
ProxyPullSupplier interface 4-3, 4-18
  connect_pull_consumer operation 4-18
ProxyPushConsumer interface 4-3, 4-17
  connect_push_supplier operation 4-18
  disconnect_push_supplier operation 4-18
ProxyPushSupplier interface 4-19
  connect_push_consumer operation 4-19
pseudo object 2-13, 3-3, 3-13, 3-18
  creating library name 3-14
publicAlgId 15-261
publicKeyBlock 15-266
PullConsumer interface 4-3, 4-10, 4-21
  disconnect_pull_consumer operation 4-7
PullSupplier interface 2-2, 4-7, 4-9
  disconnect_pull_supplier operation 4-7, 4-10
  pull operation 4-9
  try_pull operation 4-9
PushConsumer interface 2-2, 4-6, 4-8, 12-28
  disconnect_push_consumer operation 4-9
  push operation 4-8
PushSupplier interface 4-3, 4-9
  disconnect_push_supplier operation 4-7, 4-9
PV/CV delegation method 15-243

Q
QOP 15-100
QOP Policy 15-101
qualifier attributes 15-255
quality of service 2-2, 4-3, 4-4, 4-6, 4-12
query collection 11-10
query evaluator 11-3
  defined 11-19
Query Example 16-19
query framework 11-10
query framework interfaces
  overview of 11-10
Query interface
  execute operation 11-26
  get_result operation 11-27
  get_status operation 11-27
  prepare operation 11-26
  readonly attribute 11-26
query object
  defined 11-21
Query Operation 16-31
query service
  and transaction service 11-2
  list of interfaces for 11-23
Query Trader 16-71
queryable collection
  defined 11-20
QueryableCollection interface 11-25
QueryEvaluator interface
  attributes for 11-25
QueryManager interface
  create operation 11-26
Queue 17-15
Queue Interface 17-65
QueueFactory Interface 17-82

R
RACollectionFactory and RACollectionFactories Interfaces 17-74
Readonly attribute inaccuracy 14-9
Readonly attribute tdf 14-9
Readonly attribute time 14-9
Readonly attribute time_interval 14-10
Readonly attribute utc_time 14-9
received credentials 15-106
received_credentials 15-109, 15-166
received_security_features 15-110
Recipe Syntax 16-99
reclaim_message 15-168
recoverable object 10-5
  and nested transactions 10-33
recoverable server 10-6, 10-42
  glossary definition 10-88
  implementing 10-38–10-39
RecoveryCoordinator interface 10-29
  replay_completion operation 10-29
reference model xlviii
reference relationship 9-1, 9-9
  defining 9-50–9-51
  overview 9-47
refresh 15-99, 15-169
Register 16-36
register 14-17
Register Interface 16-69
Relation Interface 17-60
Relation, SortedRelation 17-13
RelationFactory Interface 17-76
relationship
  and nodes, defined 9-35
  creating 9-24
  destroying 9-26
  determining roles 9-26
relationship factory attributes 6-42, 6-44
Relationship interface 6-39, 8-25, 8-27, 9-25

copy operation 6-39
destroy operation 9-26
externalize_role operation 8-25
internalize_relationship operation 8-25
life_cycle_propagation operation 6-41
move operation 6-40
named_roles attribute 9-26
propagation_for operation 8-25
relationship service
and base level operations 9-17
and cardinality 9-2, 9-18
and containment relationship 9-47–9-48
and CORBA object references 2-10
and degree 9-2
and entity 9-2
and levels of service 9-3, 9-7–9-10
and license service 12-27
and reference relationship 9-47–9-48
and semantics 9-2
and type 9-1, 9-14
attribute and operation rationale 9-15
interface summary 9-11–9-13
overview 1-4
Relationship to Other Relevant Standards 17-131
RelationshipFactory interface 9-23
create operation 9-24
degree attribute 9-25
named_role_types attribute 9-25
relationship_type attribute 9-25
RelationshipIterator interface 9-32
destroy operation 9-32
next_n operation 9-32
next_one operation 9-32
relationships
and defining role attributes 9-30
and operations on roles 9-26–9-30
containment 8-26
reference 8-26
Remove Link Operation 16-52
Remove Type Operation 16-64
replace_audit_selectors 15-146
replace_rights 15-142
replaceable security services 15-170
Representation of Literals 16-95
representation of Time 14-1
required_rights_object 15-111
Resetting 13-19
resetting
position in an iterator 13-20
resetting position in iterator 13-20
Resetting the position in an iterator 13-19
Resolve Operation 16-45
Resource interface 10-29
commit operation 10-31
commit_one_phase operation 10-31
forget operation 10-32
prepare operation 10-30
rollback operation 10-31
resource manager 10-9, 10-68, 10-78
mappings to 10-76
resource object
defined 10-6
Restricted Access Collection Interfaces 17-64
Restricted Access Collections 17-4, 17-14
RestrictedAccessCollection Interface 17-64
restrictions 15-258
Retrieval 13-14
retrieval of PropertySet constraints 13-14
revoke_rights 15-142
rights families 15-134
RM
see resource manager
role factory attributes 6-42, 6-45
Role interface 6-37, 8-24, 9-26, 9-46
check_minimum_cardinality operation 9-29
copy operation 6-38
destroy operation 9-29
destroy_relationships operation 9-28
externalize_propagation operation 8-25
externalize_role operation 8-24
get_edges operation 9-46
get_other_related_object operation 9-27
get_other_role operation 9-27
get_relationships operation 9-28
how_many operation 9-28
internalize_role operation 8-24
life_cycle_propagation operation 6-39
link operation 9-29
move operation 6-38
related_object attribute 9-27
unlink operation 9-30
RoleFactory interface 9-27, 9-30
and max_cardinality attribute 9-31
and min_cardinality attribute 9-31
and role_type attribute 9-31
create_role operation 9-30
related_object_type attribute 9-32
roles
and cardinality 9-29, 9-31
rollback
glossary definition 10-88

S

SAId 15-247
Scoping Policies 16-13
SD interface 5-11
sealValue 15-261
SECIOP Context Management Finite State Machine Tables 15-217
SECIOP context management layer 15-212
SECIOP context management layer message header 15-212
SECIOP context management layer protocol 15-213
SECIOP sequencing layer 15-206
SECIOP tokens 15-235
when using CSI-ECMA 15-245
SECIOP-hosted CSI protocols 15-224
secretAlgId 15-262
secure association options 15-149
secure DCE CIOP 15-306
secure DCE-CIOP operational semantics 15-280
secure interoperability bridges 15-177
secure interoperability levels 15-183

Index

secure inter-ORB protocol (SECIOP) 15-301
secure invocation 15-63, 15-194
secure invocations
  implementor's view of 15-79
Secure Object Creation
  Implementor's View of 15-85
secure object invocations 15-21
Secure Time 14-18
secure_universal_time 14-12, 14-17
SecureUniversalTime 14-3
security association 15-174
security associations
  establishing 15-22
security attributes
  selecting 15-61
Security Context 15-83
Security Context object 15-165
Security Features Policy 15-102
security functionality packages 15-89
security mechanism types 15-175
Security Mechanisms 15-100
security mechanisms 15-67, 15-309
  hosted directly on IIOP 15-185
  hosted on SECIOP 15-184
security model
  administrator's view of 15-48
  application developer's view of 15-47
  end user's view of 15-47
  enterprise management view of 15-46
  object system implementor's view of 15-48
security name 15-227
security reference model
  definition of 15-18
Security Service 15-1
security service
  and license service 12-27
security specification 15-3
security technology 15-55
security_features 15-166
securityValue 15-254
see also data objects
sending Time across the network 14-23
seq-number 15-248, 15-249, 15-251
Sequence 17-13
Sequence Interface 17-63
SequencedDataFrame
  sending a 15-211
SequenceFactory Interface 17-80
SequentialCollection Interface 17-31
serialNumber 15-257
Service Offers 16-7
Service Type Repository 16-59
Service Type Repository Module 16-88
set 13-18
Set, SortedSet 17-13
set_association_options 15-150
set_audit_channel 15-147
set_audit_selectors 15-145
set_credentials 15-108
set_data 14-16
set_delegation_mode 15-152
set_NR_features 15-124
set_NR_policy_info 15-155
set_policy_override 15-100
set_policy_overrides 15-104
set_privileges 15-97
set_required_rights 15-135
set_security_features 15-95
set_timer 14-16
SetFactory Interface 17-76
signature 15-267
signatureValue 15-261
signedPKBPart 15-266
simple name 3-2
Simple Trader 16-72
SNA LU protocol 10-80, 10-82
  incoming communication 10-83
  outgoing communication 10-83
  transaction identifiers 10-82
SortedBag Interface 17-63
SortedCollection Interface 17-37
SortedIterator Interface 17-110
SortedMap Interface 17-62
SortedMapFactory Interface 17-78
SortedRelation Interface 17-62
SortedRelationFactory Interface 17-79
SortedSet Interface 17-63
SortedSetFactory Interface 17-79
source of Time 14-2
spans 14-11
SPKM
  using for SECIOP 15-233
SPKM protocol 15-231
SPKM_REQ Used in Public Key Mechanism
  Profile of 15-270
SQL Query Language 11-6
SSL 15-306
Stack 17-15
Stack Interface 17-67
StackFactory Interface 17-82
Stand-alone Trader 16-72
standard data types
  values for 15-307
Standardized Capability Supported Policies 16-15
Stream interface 8-13, 8-14
  begin_context operation 8-15
  end_context operation 8-15
  externalize operation 8-14
  flush operation 8-15
  internalize operation 8-14, 8-15
  internalize_from_stream operation 8-16
stream object
  creating 8-14
  data format 8-29–8-31
  externalizing 8-14
  externalizing group 8-15
  internalizing 8-14, 8-15
stream service 8-4
  and begin_context request 8-4
  and externalize_to_stream request 8-4
  and internalize_from_stream request 8-4
  and readonly key attribute 8-4

Streamable interface 8-4, 8-8, 8-21
externalize_to_stream operation 8-21
internalize_from_stream 8-22
is_identical operation 8-21
streamable object
and inheritance 8-21
creating
create_uninitialized operation 8-23
creation key 8-21
StreamableFactory interface 8-23
StreamFactory interface 8-9, 8-13, 8-14
create operation 8-14
StreamIO interface 8-4, 8-9, 8-20
read_ operation 8-22
read_object operation 8-22
read_t operation 8-20
write_ operation 8-21, 8-30
write_object operation 8-21
write_operation 8-20
structural model 15-49
SubtransactionAwareResource interface 10-33
commit_substransaction operation 10-33
commit_subtransaction operation 10-33
subtransactions 10-7, 10-12, 10-56, 10-58, 10-60, 10-62, 10-66
subtyping 2-1, 2-5
supplier 4-2
SupplierAdmin interface 4-3, 4-16, 4-17
for_suppliers operation 4-16
obtain_pull_consumer operation 4-17
obtain_push_consumer operation 4-17
SupportAttributes 16-29
symmetric key distribution with asymmetric KDS 15-240
symmIntradomain scheme 15-264
synchronization of Time 14-18
synchronized data interface
see SD interface

T
TAG_ASSOCIATION_OPTIONS 15-180, 15-276
TAG_DCE_SEC_MECH 15-275
TAG_GENERIC_SEC_MECH 15-179
TAG_SEC_NAME 15-181, 15-278
TAG_x_SEC_MECH 15-179
target
security at the target 15-64
target_requires field 15-277
target_supports field 15-277
targetAddress 15-248
TargetAEF Part 15-248
targetAEFPart 15-247
targetAEFPartSeal 15-247
targetIdentity 15-249
targetKeyBlock 15-249
targetName 15-262, 15-267
TargetResultToken 15-229, 15-246, 15-249
Terminator interface
rollback operation 10-23
terminator object 10-41
time 14-11
Time Interval Object (TIO) 14-10
Time Interval Objects (TIOs) 14-3
Time Service interface 14-11
Time Service Requirements 14-1
Time Service requirements 14-1
time_set 14-16
time_to_interval 14-10
TimeBase 14-4, 14-5
timePeriods 15-259
Timer Event Handler 14-3, 14-15
Timer Event Service 14-3, 14-4, 14-13, 14-16, 14-22
TimeUnavailable 14-4, 14-8
tokenId 15-247
tokentype 15-214
Trader Attributes 16-21
Trader Policies 16-16
trading object service 16-2
transacations
resource manager 10-68
transaction abort
see Resource interface
rollback operation 10-31
transaction context 10-18
management of 10-22
propagation of 10-22
transaction originator 10-13, 10-19, 10-23, 10-47
glossary definition 10-90
transaction service
and concurrency control service 2-10
and orb interoperability 2-12
and persistent object service 2-11
application use of 10-34
transactional client 10-4, 10-38
glossary defintion 10-89
transactional object 10-4
example 10-43
transactional server
defined 10-6
TransactionalLockSet interface 7-9
TransactionalLockSet interface operations 7-12
TransactionalObject interface 10-34
TransactionFactory interface 10-41
transactions
checked 10-36–10-38, 10-39, 10-40
consistency property 10-58
consistency property,glossary definition 10-86
coordinator object 10-30, 10-33, 10-41, 10-42, 10-53, 10-61
distributed 10-40
durability 10-56
durability, glossary definition 10-86
flat 10-6, 10-7, 10-9, 10-39
flat,glossary definition 10-86
implicit propagation 10-41
interposition 10-49, 10-60, 10-62
interposition, glossary defintion 10-87
isolation 10-7, 10-9, 10-13, 10-23
isolation, glossary definition 10-87
propagation 10-34–10-38, 10-41, 10-44, 10-60, 10-63, 10-65, 10-66, 10-88
propagation to resource manager 10-78
recoverable object 10-5, 10-33
recoverable server 10-6, 10-38–10-39
recoverable server, glossary defintion 10-88

Index

recoverable server,example 10-42
resource manager 10-9, 10-78
terminator object 10-41
two-phase commit protocol 2-11, 10-12, 10-29, 10-49, 10-52, 10-58, 10-62, 10-68, 10-80, 10-83
two-phase commit, glossary definition 10-90
TraveralCriteria interface
next_n operation 9-44
traversal criteria
creating 6-41, 9-36
example of 9-37
Traversal interface
destroy operation 9-43
next_n operation 9-43
next_one operation 9-42
ScopedEdge structure 9-42
traversal object 9-35, 9-36
creating 9-41
TraversalCriteria interface 9-36, 9-43
destroy operation 9-44
next_one operation 9-43
visit_node operation 9-44
Weighted_Edge structure 9-43
TraversalFactory interface 9-41
create_traversal_on operation 9-42
trtContents 15-249
trtSeal 15-250
Type checking information 17-22
Type Definitions 17-19
Type InaccuracyT 14-6
Type IntervalT 14-6
Type safety 17-7
Type TdfT 14-6
Type TimerEventT 14-15
Type TimeT 14-6
Type UtcT 14-6
TypedConsumerAdmin interface
obtain_typed_pull_supplier operation 4-26
obtain_typed_push_supplier operation 4-26
TypedProxyPullSupplier interface 4-28
TypedProxyPushConsumer interface 4-27
TypedPullSupplier interface 4-21
TypedPushConsumer interface 4-20
TypedSupplierAdmin interface 4-27
obtain_typed_pull_consumer operation 4-27
obtain_typed_push_consumer operation 4-27

U
Unique entries (collections) 17-4
universal object identity 2-5
Universal Time Coordinated (UTC) 14-1
Universal Time Object (UTO) 14-8
Universal Time Objects (UTOs) 14-3
universal_time 14-4, 14-12
UniversalTime 14-3
Unmask Type Operation 16-66
unregister 14-17
useAlgId 15-272
usec 15-248
useHashAlgId 15-272
user sponsor 15-59
userData 15-251
UserEnvironment interface
operations 5-37
utcTime 15-248
uto_from_utc 14-12

V
validity 15-257
Vault 15-83
Vault object 15-162
verify_evidence 15-126
void destroy() 15-95

W
Withdraw Operation 16-41
Withdraw Proxy Operation 16-58
Withdraw Using Constraint Operation 16-44
WrapToken 15-230, 15-246, 15-252

X
X/Open xlviii
X/Open CLI standard 5-34
X/Open TX interface 10-74–10-76
X/Open XA interface 10-68

CORBAservices: Common Object Services Specification

TO: *CORBAservices* Readers

FROM: OMG Headquarters

RE: Security Update package for Security Service Specification v/1.2

DATE: December 21, 1998

Pages to remove from CORBA services	**Pages to add from this update**
Title and copyright	Title and copyright
Table of Contents (footer reads November 1997)	Table of Contents (footer reads December 1998)
List of Figures (footer reads November 1997)	List of Figures (footer reads December 1998)
List of Tables (footer reads November 1997)	List of Tables (footer reads December 1998)
Chapter 8 - Externalization Service (footer reads March 1995)	Chapter 8 - Externalization Service (footer reads December 1998)
Chapter 15 - Security Service (footer reads November 1996)	Chapter 15 - Security Service (footer reads December 1998)
Index (footer reads November 1997)	Index (footer reads December 1998)