Le standard CORBA

Frank Singhoff

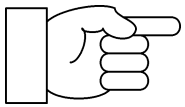
Bureau C-203

Université de Brest, France

Laboratoire Lab-STICC UMR CNRS 6285

singhoff@univ-brest.fr

Introduction



CORBA (pour **C**ommon **O**bject **R**equest **B**roker **A**rchitecture) est un standard normalisant l'accès à des objets répartis avec des objectifs :

- Via un **middleware, ou intergiciel**.
- D'interopérabilité, de portabilité et de réutilisabilité.
- De transparence à la localisation. Pas de transparence d'accès.

et définissant principalement :

- Une architecture standard (réutilisabilité).
- Un langage de description des objets (IDL, pour **I**nterface **D**efinition **L**anguage) ainsi que sa correspondance avec le langage utilisé pour l'implantation des objets.
- Un protocole assurant l'interopérabilité.

Historique

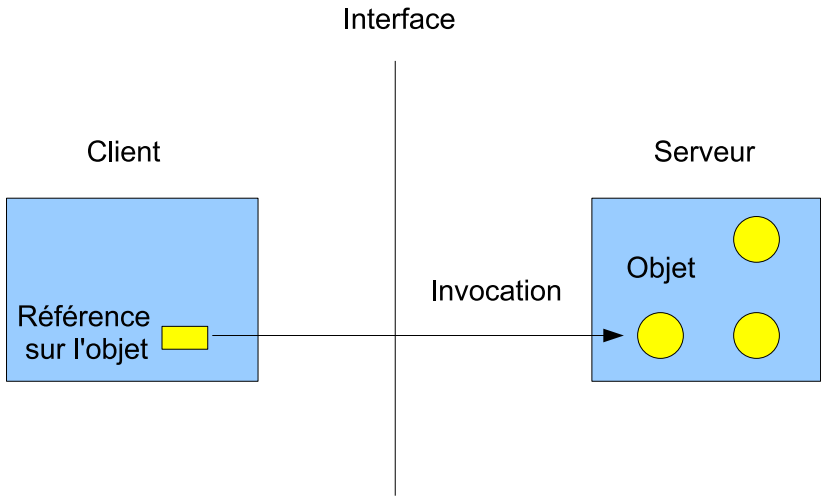
Standard défini par l'OMG [a].

- Consortium créé en 1989.
- www.corba.org
- Historiques standards : Corba 1.0 (1991 [OMG 93]), Corba 2.0 à 2.6 (1995-2006 [OMG 99]), Corba 3.3 (2002-2012), Corba 3.4 (2021).
- Exemples de produits : VisiBroker, Orbix, JacORB ou TAO de Microfocus (https://www.microfocus.com/).

[a] **OMG** pour **O**bject **M**anagement **G**roup.

Sommaire

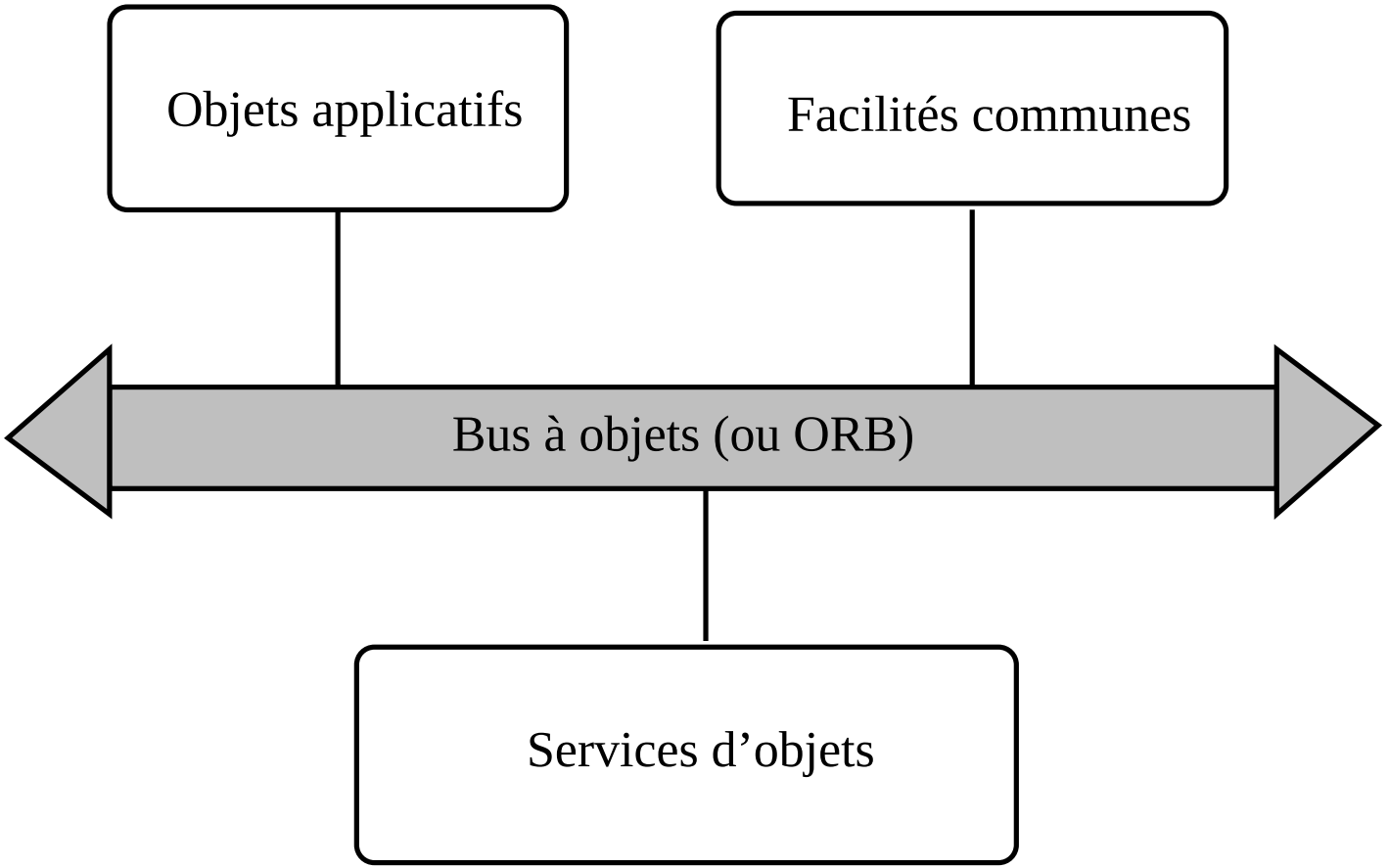
Le modèle objet de CORBA



- **Notions de :**
  - Objets, interfaces, opérations, implémentations, références d'objet.
  - Séparation de l'interface et de son implémentation. Le client ne connait que l'interface.
  - Requêtes (référence + opération + paramètres + exceptions + contexte). Mécanisme synchrone.
  - Sémantique d'invocation (exactement une fois ou au plus une fois).

L’architecture OMA (1)

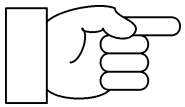
OMA pour **O**bject **M**anagement **A**rchitecture.



L’architecture OMA (2)

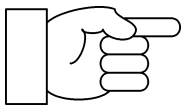
- ORB (Object Request Broker) ; ou bus à objets. Fournit les fonctionnalités nécessaires à l’acheminement des invocations.
- Les services d’objets fournissent des fonctionnalités de bas niveaux.
- Les objets applicatifs sont les objets développés par les utilisateurs de l’architecture.
- Les facilités sont des objets intermédiaires entre les services et les objets applicatifs.

Et concrétement (1)



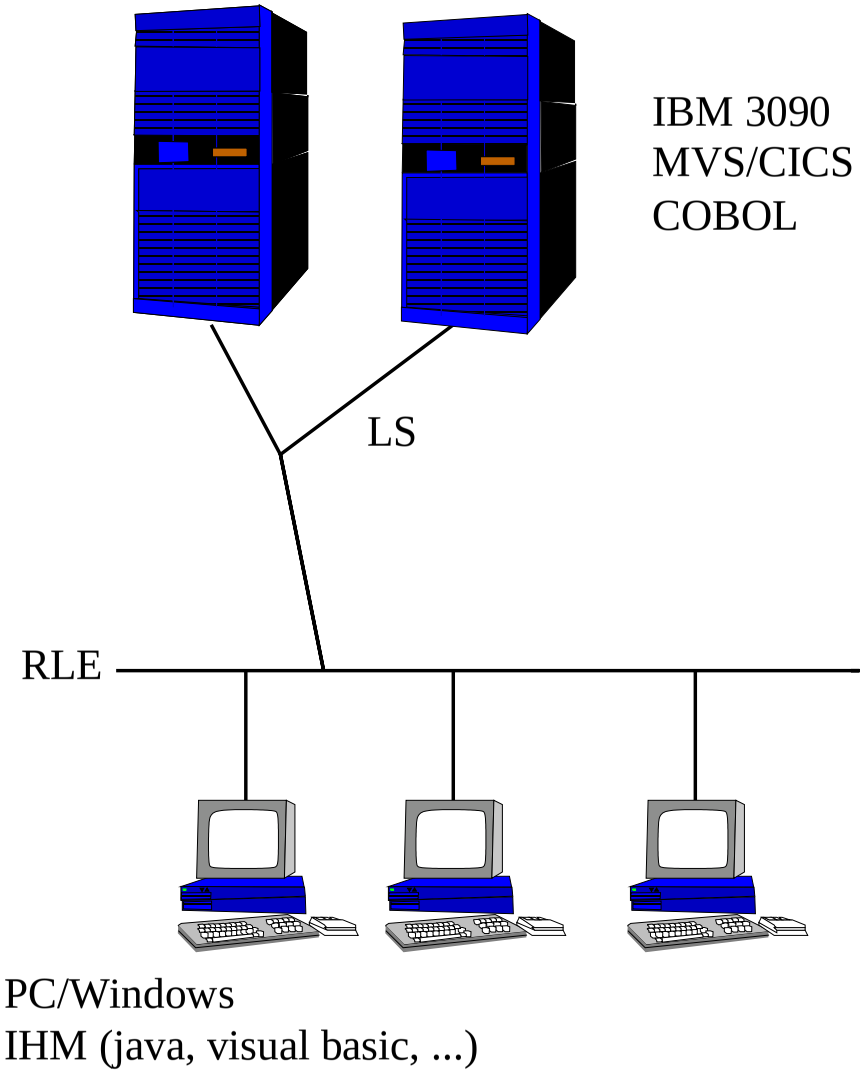
Un ORB peut être :

- Une simple bibliothèque.
- Un ou des processus.
- Un système d'exploitation.
- Etc.

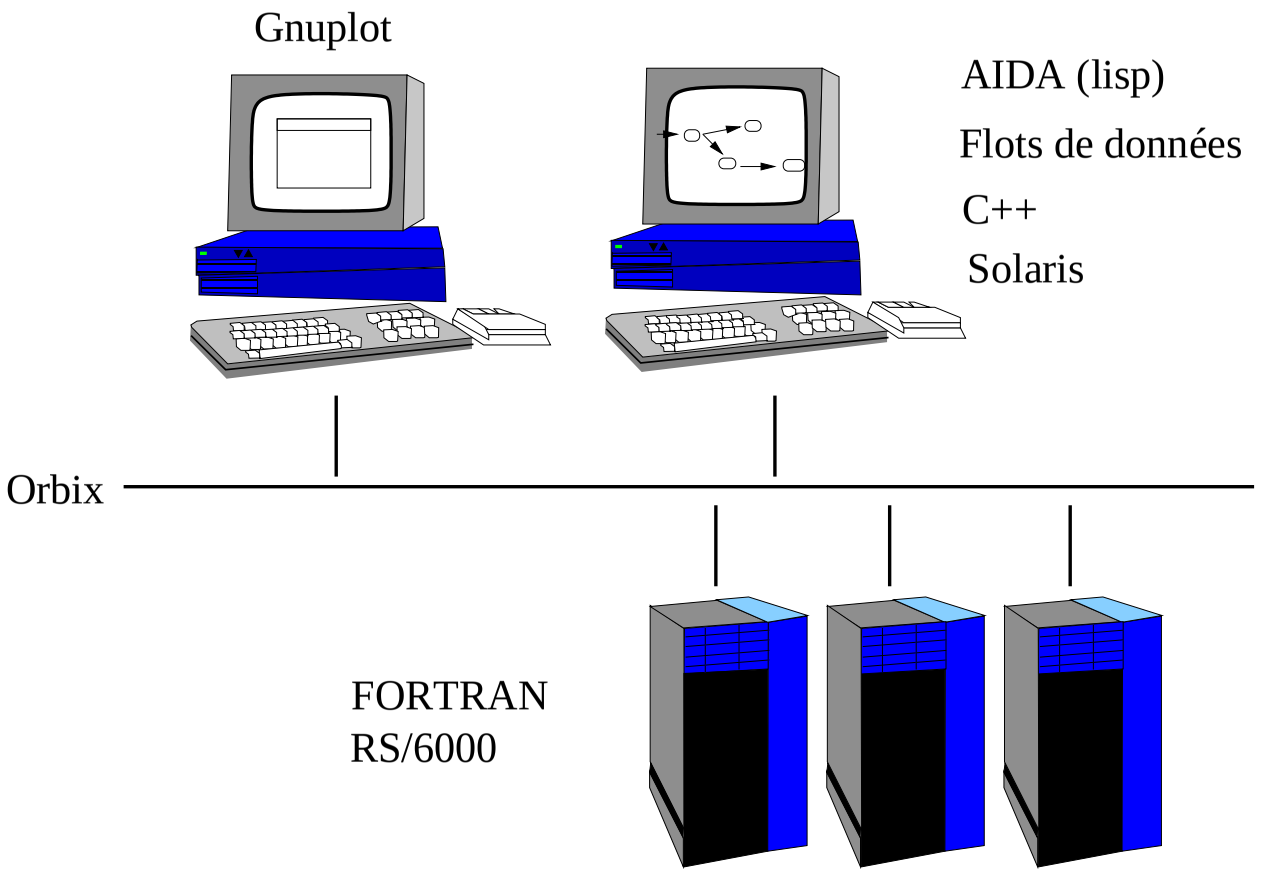


Un client, un objet, un serveur peuvent être ...

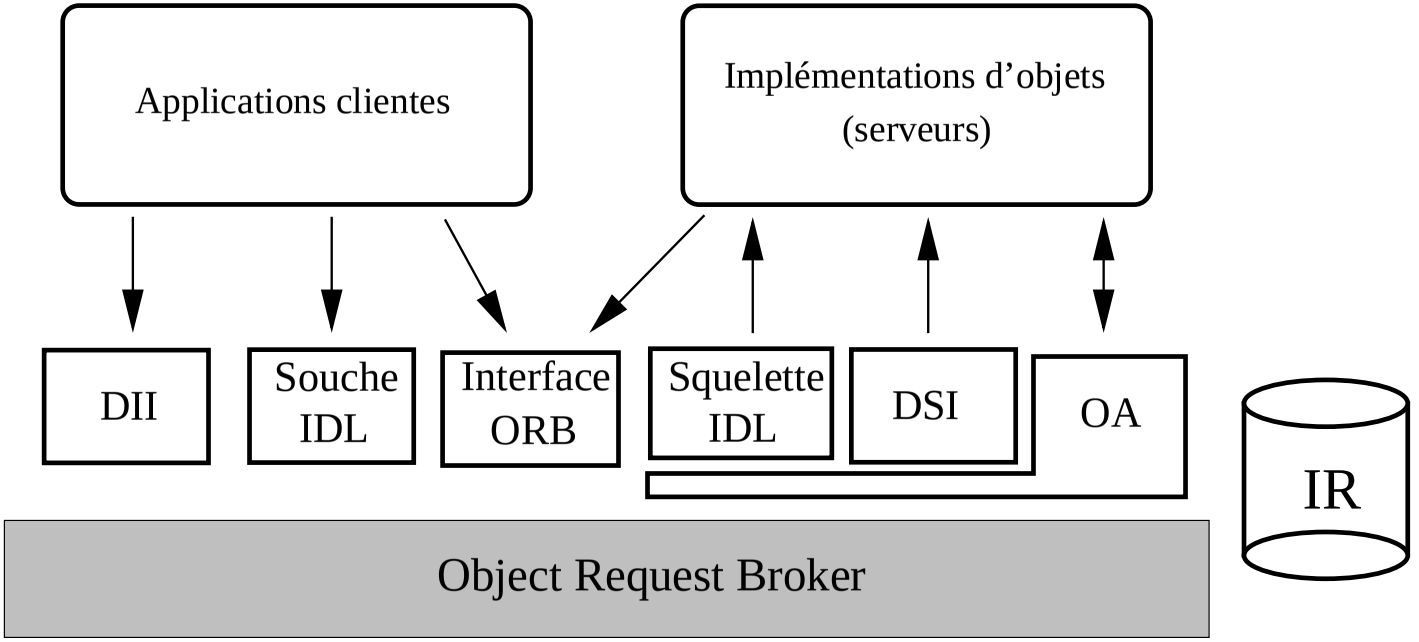
Et concrétement (2)



Et concrétement (3)



Structure d'un bus à objets (1)



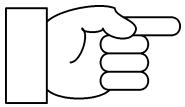
Structure d’un bus à objets (2)

- Interface de l’ORB. Offre des services de base communs aux clients et aux serveurs.
- Souche et squelette IDL. Invocation statique des objets.
- OA (Object Adapter). Gestion des implémentations d’objets et des serveurs.
- DII (Dynamic Invocation Interface). Interface d’invocation dynamique.
- DSI (Dynamic Skeleton Interface). Squelette dynamique d’invocation.
- Interface Repository. Référentiel des interfaces d’objets.
- Implementation Repository. Référentiel des implémentations d’objets.

Sommaire

1. Concepts de base et architecture.
2. Le langage de description d'interface.
3. Zoom sur le bus à objets.
4. Mapping vers C++ et Java.
5. Les services et les facilités CORBA.
6. CORBA et l'interopérabilité.
7. Conclusion.
8. Références.

L'IDL : présentation (1)



Objectif : permettre une description des objets de façon indépendante du langage d'implantation et du système sous-jacent.

- *Dans CORBA, tout est IDL !* Notion d'objets et de pseudo-objets.
- Très proche du C++.
- CORBA définit les règles de correspondance de l'IDL vers les langages cibles (*mapping*). Mapping normalisé pour C, C++, Ada, Smalltalk, COBOL, Java.
- Utilisation de préprocesseurs possible. Invocation statique $\Longrightarrow$ compilation pour générer les souches et squelettes.

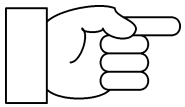
L'IDL : présentation (2)

- **Un mapping ? pourquoi faire ?** $\Longrightarrow$ API portable

- **Difficultés :**
  - Abstractions offertes par le langage ciblé (ex : COBOL, C) et particularités fonctionnelles.
  - Portabilité du langage (ex : C++).

- **Ce que normalise le mapping :**
  - Spécifications IDL des objets (types, exceptions, constantes, etc).
  - Pseudo objets CORBA (ORB, BOA, DII, etc).
  - Les souches et squelettes.

L'IDL : présentation (3)



Ce qui est défini en IDL :

- Des modules. Unités de structuration (opérateur de portée ::).
- Des constantes.
- Des interfaces.
- Des opérations.
- Des types construits à partir de types simples prédéfinis.
- Des exceptions utilisateurs. Les objets peuvent lever des exceptions et retourner des informations.

Les interfaces

Unité représentant un point d'accès à un objet. Héritages multiples sous certaines contraintes (ex : pas de surcharge de méthodes).

Est constituée de :

- Types, exceptions.
- Attributs (option *readonly*).
- Opérations.

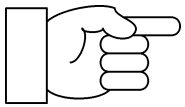
Les opérations

☞ Opération = identificateur + paramètres + valeur de retour + sémantique d'invocation + exceptions + contexte :

- Paramètres *in*/*out*/*inout*.
- Sémantique d'invocation. Méthodes *oneway* = au plus une fois. Exactement une fois sinon (sauf si exception levée : au plus une fois dans ce dernier cas).

! *oneway* $\neq$ invocation asynchrone.

Types simples et construits (1)



Types simples :

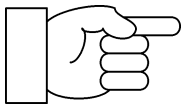
- *long, short, char, double, unsigned, boolean*.
- *string*. Chaîne de caractères bornée ou non.
- *octet*. Information de 8 bits sans conversion.
- *Object*. Désigne une référence d'objet CORBA.
- *any*. Désigne n'importe quels types IDL.

$\Longrightarrow$ Extensions mineures à partir de CORBA 2.1

Exemple 1

```
module banque {
   interface compte;
   interface compte {

      readonly attribute long numero_compte;
      attribute string titulaire;
      readonly attribute double solde;

      void nombre_operations(out long nombre);
      void debiter(in double montant);
      void crediter(in double montant);
      void prelevement(in double montant, in compte dest);
      oneway void archivage(in string date);
   };
};
```

Types simples et construits (2)



Constructeurs de type :

- opérateur *typedef*.
- *struct*. Structure, identique au langage C/C++.
- *enum*. Énumération ordonnée.
- *union*.
- *sequence* et tableaux. Séquence = tableau à une dimension d'un type quelconque. Séquence bornée ou non. Notion de taille maximale et de taille courante.
- *interface*. Objet fils de *Object*.

Exemple 2

```
module file_system {

   interface file {
      long read(inout string data);
      long write(in string data);
      long close();
      long ioctl(in long param);
   };

   struct stat_data {
      string device;
      long   inode;
      long   uid;
      long   gid;
   };
...
```

Exemple 2

```
   ...
   exception error {long error_code;};
   typedef  sequence<string> name_table;
   enum mode {read_only, write_only, read_write};

   interface vfs {
      file open(in string name, in mode m)
         raises(error);
      stat_data stat(in string name);
      long chmod(in string name, in long permissions)
         raises(error);
      long chmod_list(in name_table name, in long perm)
         raises(error);
      oneway void sync();
   };
};
```

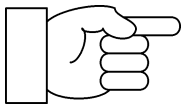
Sommaire

1. Concepts de base et architecture.
2. Le langage de description d'interface.
3. Zoom sur le bus à objets.
4. Mapping vers C++ et Java.
5. Les services et les facilités CORBA.
6. CORBA et l'interopérabilité.
7. Conclusion.
8. Références.

Zoom sur le bus à objets

1. Les services du noyau de l'ORB.
2. Les adaptateurs d'objets et leurs interactions avec les serveurs.
3. Invocation statique : souches et squelettes.

Services du noyau de l'ORB (1)



Fonctions communes aux clients et aux serveurs, et qui concernent :

- L'initialisation de l'environnement CORBA.
- La gestion des références d'objets.
- L'accès aux objets qui fournissent des services de base (service de nom, référentiel d'interface).
- La gestion des pseudo-objets (ex : *NVList, Request, Context, Environnement, NamedValue, ORB, OA, TypeCode, Principal*).

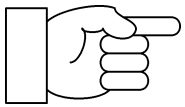
Services du noyau de l'ORB (2)

```
module CORBA {
  ORB ORB_init(in string argv, in ORBid orb);

  pseudo interface ORB {
    string object_to_string(in Object obj);
    Object string_to_object(in string str);
    void run();
    void shutdown(in boolean b)
    Object resolve_initial_references(in Objectid ident);
    create_{NVList, NamedValue, Context};
  };
  pseudo interface Object {
    boolean is_nil();
    boolean is_a();
    Object duplicate();
    void release();
    interfaceDef get_interface();
    implementationDef get_implementation();
    create_Request(...);
  };
};
```

Zoom sur le bus à objets

1. Les services du noyau de l'ORB.
2. Les adaptateurs d'objets et leurs interactions avec les serveurs.
3. Invocation statique et dynamique.

Les adaptateurs d'objets (1)



Fonctions d'un adaptateur = protocole ORB/Serveurs :

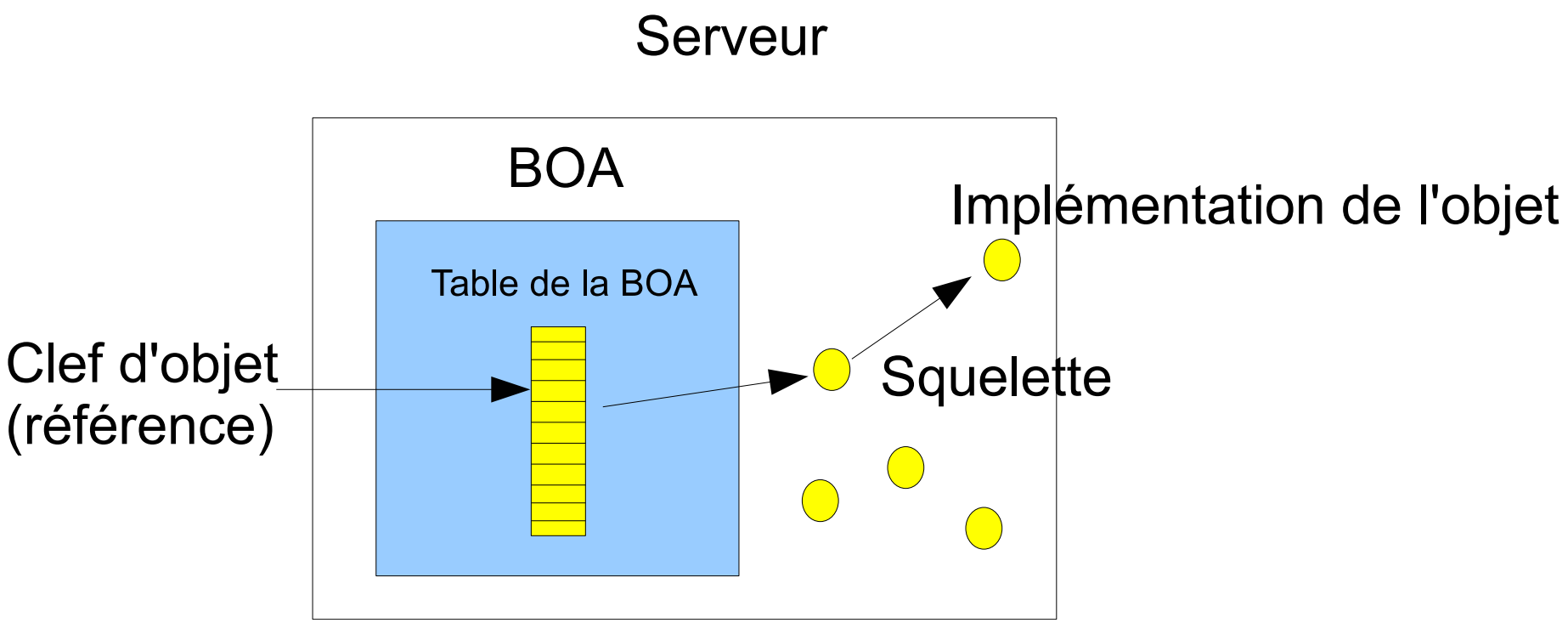
- Gestion des implémentations d'objets et des serveurs. Serveurs $\neq$ objets.
- Activation, désactivation et invocation des objets.
- Sécurité.

$\Longrightarrow$ Historique : BOA [a] puis POA [b].

[a] **BOA** pour **B**asic **O**bject **A**daptor.

[b] **POA** pour **P**ortable **O**bject **A**daptor.

Les adaptateurs d’objets (2)

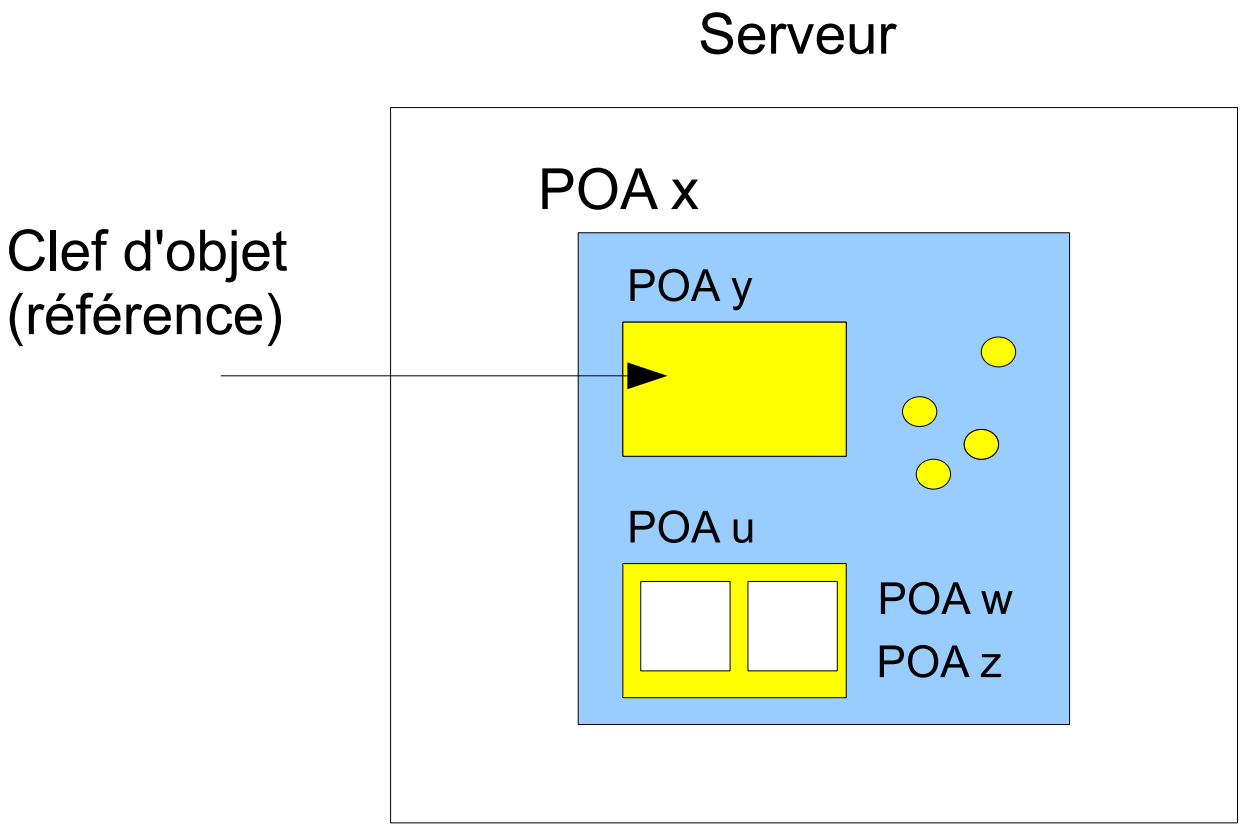


- Applications peu portables car spécifications peu précises : pas de notion précise de *thread*, de serveurs, de mode d’activation.
- Choix de la politique d’activation laissé aux implanteurs d’ORB.

Les adaptateurs d'objets (3)

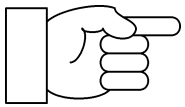
```
module CORBA {
  pseudo interface BOA {

    //Object activation/deactivation
    //
    void obj_is_ready(in Object obj,
        in ImplementationDef impl);
    void deactivate_obj(in Object obj);

    //Server activation/deactivation
    //
    void impl_is_ready(in ImplementationDef impl);
    void deactivate_impl(in ImplementationDef impl);
  ...
  }
}
```

Les adaptateurs d'objets (4)



- Notion de politique : un POA = une politique donnée.
- Doit permettre une meilleure portabilité des serveurs : un serveur nécessite alors une POA dont la spécification est connue.

Les adaptateurs d’objets (5)



Exemples de politique :

- Modèle de thread.
- Comportement lors d’une invocation (recherche de l’objet, filtre).
- Persistence, mode d’activation des objets et serveurs (automatique ou non, objet de courte durée de vie), choix de l’exécutant (pool, gestion de ressource).
- Gestion des POA (structuration de l’application).
- ...

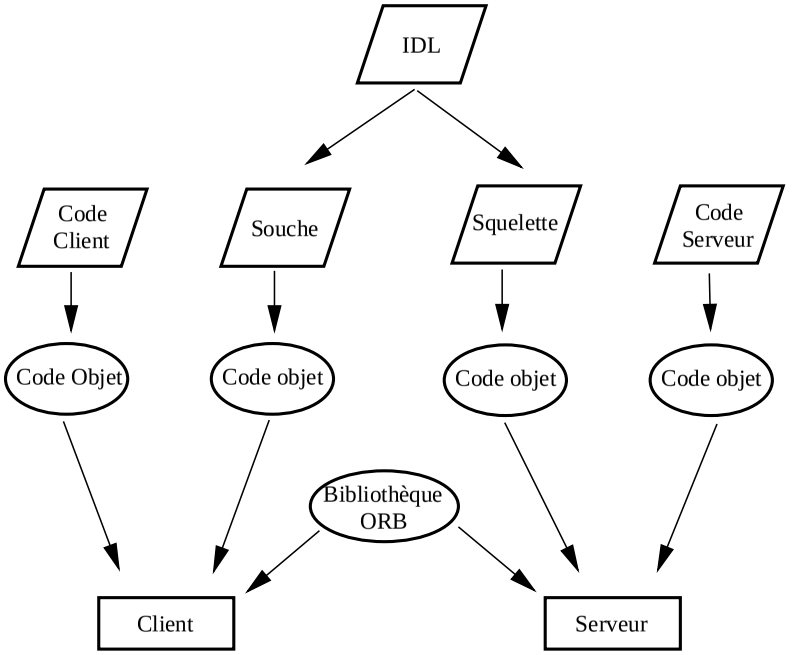
Les adaptateurs d'objets (6)

```
module CORBA {
  pseudo interface POA {

  // POA creation and destruction
  POA create_POA(in string adapter_name,
      in POAManager a_POAManager,
      in CORBA::PolicyList policies);
  POA find_POA(in string adapter_name,
      in boolean activate_it);
  void destroy(in boolean etherealize_objects,
       in boolean wait_for_completion);

  // object activation and deactivation
  ObjectId activate_object(in Servant p_servant);
  void activate_object_with_id (in ObjectId id,
       in Servant p_servant);
  void deactivate_object(in ObjectId oid);
  Object servant_to_reference (in Servant p_servant);
...
}
```

Zoom sur le bus à objets

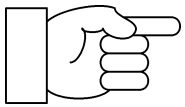
1. Les services du noyau de l'ORB.
2. Les adaptateurs d'objets et leurs interactions avec les serveurs.
3. Invocation statique et dynamique.

Invocation statique (1)



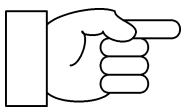
- Interfaces des objets connues à la compilation. Contrôle de type.
- Certainement la solution la plus utilisée, car simple et efficace. Invocation d'un objet CORBA identique à un objet quelconque.

Invocation statique (2)



Fonctions des souches :

- Emballage des invocations.
- Transmission vers le serveur.

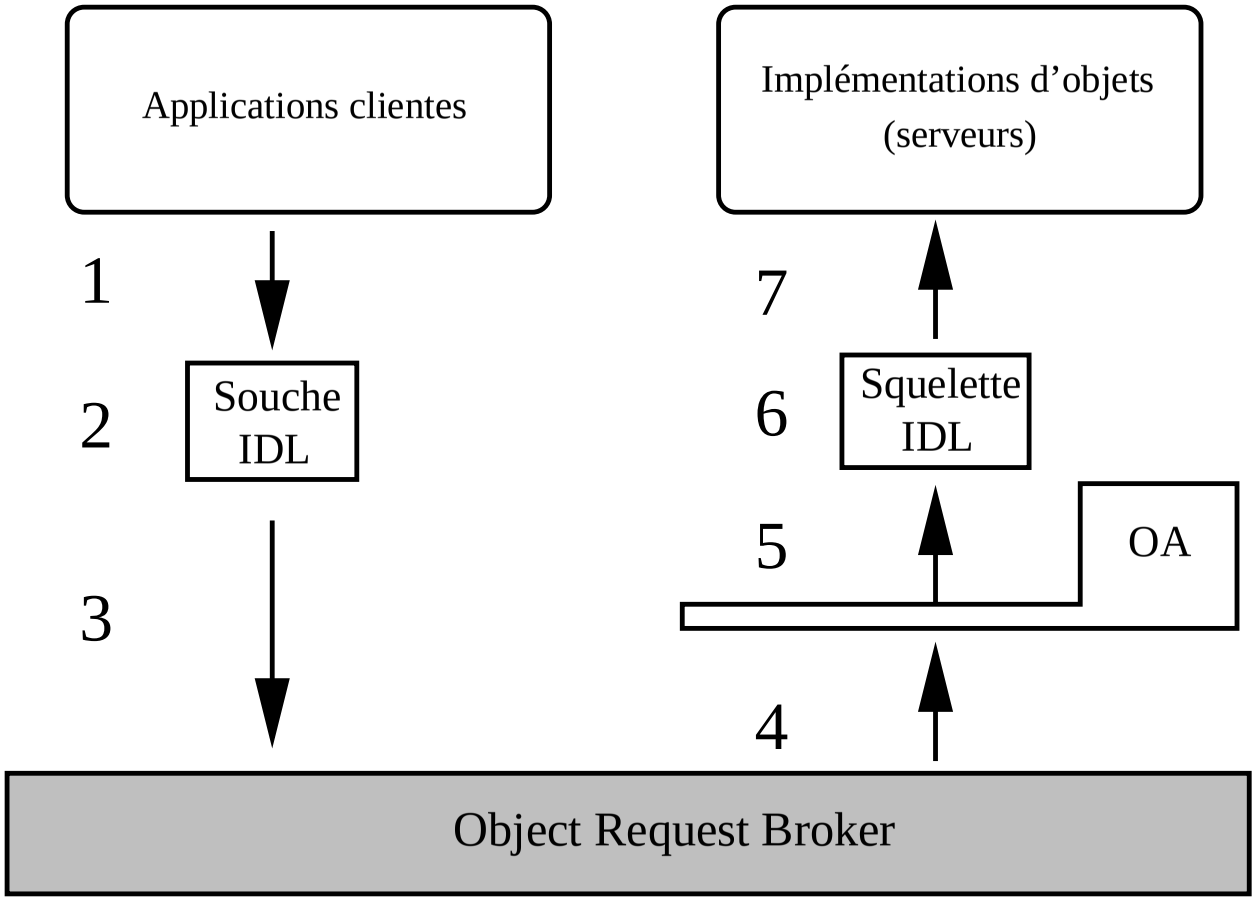


Fonctions des squelettes :

- Déballage des invocations.
- Permettre aux OA d'invoquer les méthodes concernées (un squelette fournit les *upcalls* vers l'implémentation).

Invocation statique (3)

- Cheminement d'une invocation :



Invocation statique (4)

1. Invocation d’une méthode par un client.
2. L’objet invoqué (la souche) emballe les paramètres, et construit un message.
3. Grâce à la référence, l’ORB transmet la requête vers le site du serveur.
4. L’OA réceptionne la requête.
5. Grâce à la clef d’objet, l’OA transmet la requête au squelette.
6. Le squelette déballe les paramètres,
7. Le squelette invoque l’implémentation.

Invocation statique (5)

- Structure d’un client :

1. Initialisation de l’environnement CORBA.
2. Récupération d’une référence d’objet sur le service de désignation.
3. Obtention d’une souche sur le service de désignation.
4. Récupération d’une référence d’objet sur le service à invoquer.
5. Obtention d’une souche sur le service à invoquer.
6. Invocations.

Invocation statique (6)

- Structure d’un serveur :

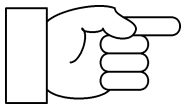
1. Initialisation de l’environnement CORBA.
2. Création d’une instance d’implémentation.
3. Activation de l’instance : déclaration à l’OA.
4. Récupération d’une référence d’objet sur le service de désignation.
5. Obtention d’une souche sur le service de désignation.
6. Publication du service par le service de désignation.
7. Démarrage de l’OA.

Sommaire

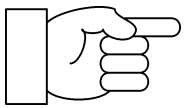
Sommaire

1. Mapping vers C++.
2. Mapping vers Java.

Généralités (1)



Objectif d'un mapping : définition de l'API manipulée par le développeur lors de la construction d'applications CORBA *portables*.



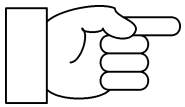
Le mapping C++ est défini dans CORBA 2. Problèmes :

- Gestion mémoire de C++.
- Incompatibilités entre les différents compilateurs (RTTI [a], gestion des exceptions, *namespace*).

$\Longrightarrow$ le mapping offre sur certains points plusieurs alternatives $\Longrightarrow$ pb de portabilité.

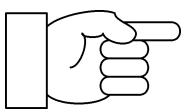
[a] **RTTI** pour **R**un-**T**ime **T**ype **I**nformation.

Généralités (2)



Ce que définit le mapping C++ :

- Signatures des classes qui vont implanter les squelettes et les souches.
- Correspondance type IDL vers type C++.
- Correspondance pseudo-objets en C++ $\Longrightarrow$ une classe C++ par pseudo-objet décrivant les prototypes des opérations spécifiées en “pseudo” IDL.

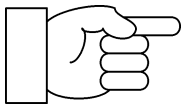


La classe CORBA regroupe la définition des pseudo-objets et des types standards.

Mapping des types IDL (1)

- Types standards *double, long, octet*, etc. = *CORBA::Double*, *CORBA::Long*, etc.
- Types construits *sequence, string, union* = classe C++. Ces classes exportent des méthodes permettant de manipuler certaines caractéristiques des types IDL.
- Mapping pour les interfaces du coté client : héritage de *CORBA::Object*. Ajout de certaines fonctionnalités telles que *narrow*.

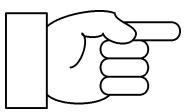
Mapping des types IDL (2)



Mapping pour les opérations :

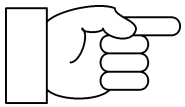
- La gestion mémoire entre l'appelant et l'appelé.
- Règles de passage des paramètres ainsi que la signature des méthodes.

- Ces règles dépendent :
  - Du type des paramètres manipulés (de taille fixe ou variable).
  - Si les paramètres sont passés en *in*,*out* ou en *inout*.



Mapping pour les attributs : par deux méthodes, une seule si l'attribut est *readonly*.

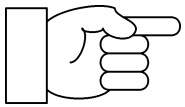
La gestion mémoire



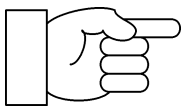
Pb : le mapping introduit de nombreuses allocations dynamiques (pour l'utilisation de certains types IDL).

- Le mapping offre des outils permettant une gestion automatique de l'allocation et de la libération de la mémoire :
  - Par les types *T_ptr* et *T_var*.
  - Par l'ajout de mécanismes dans certaines classes implantant des types IDL (*sequence*, *union*, *string*, etc.). Lors d'affectation, de sortie de blocs.
- La gestion automatique peut parfois être désactivée (ex : *sequence*).

Les classes d'implémentation



Du coté serveur, une interface IDL est implantée par une classe C++.



CORBA **propose** deux méthodes pour connecter une classe d'implémentation à son squelette, tout en laissant à l'implanteur une totale liberté :

- Par héritage. Le squelette est une classe abstraite qui fournit une méthode virtuelle pure pour chaque opération IDL. La classe d'implémentation est fille du squelette.
- Par la délégation (ou *tie*). La classe *tie* joue le rôle de squelette et invoque directement la classe d'implémentation.

Invocation statique en C++

```
CORBA::ORB_ptr orb = CORBA::ORB_init();

CORBA::Object_var obj;
CORBA::Object_var sn;

sn = orb->resolve_initial_references("NameService");

NamingContext_var nom = NamingContext::_narrow(sn);
if (CORBA::is_nil(nom))
  fprintf(stderr,"Echec lors du narrow\n");

obj = nom->resolve("nom du service");

compte_var cpt = compte::_narrow(obj);
if (CORBA::is_nil(cpt))
  fprintf(stderr,"Echec lors du narrow\n");

solde=cpt->debiter(1000);
```

Invocation dynamique en C++

```
CORBA::ORB_ptr orb = CORBA::ORB_init();

CORBA::Object_var obj;
CORBA::Object_var sn;

sn = orb->resolve_initial_references("NameService");

NamingContext_var nom = NamingContext::_narrow(sn);
if (CORBA::is_nil(nom))
  fprintf(stderr,"Echec lors du narrow\n");

obj = nom->resolve("nom du service");

CORBA::Request_var request;
request = obj -> _request("debiter");
request -> add_in_arg() <<= (CORBA::Long) 1000 ;
request -> invoke();
solde = request -> extract_out_arg();
```

Exemple d’un serveur CORBA

```
CORBA::ORB_ptr orb = CORBA::ORB_init();
CORBA::BOA_ptr boa = orb->BOA_init();

compte_impl *cpt = new compte_impl;

boa->obj_is_ready(cpt);

CORBA::Object_var sn;
sn = orb->resolve_initial_references("NameService");

NamingContext_var nom = NamingContext::_narrow(sn);
if (CORBA::is_nil(nom))
  fprintf(stderr,"Echec lors du narrow\n");

nom->bind(cpt,"nom du service");

boa->impl_is_ready();
```

Sommaire

1. Mapping vers C++.
2. Mapping vers Java.

Objet, référence, souche et squelette

- Référence CORBA : classe org.omg.CORBA.Object.
- Interface IDL : une interface Java + une classe $interfaceHelper$ et $interfaceHolder$.
- Souche IDL : une classe Java avec héritage de org.omg.CORBA.Object.
- Squelette IDL : une classe Java (liens d'héritage implementation $\Longrightarrow$ squelette $\Longrightarrow$ org.omg.CORBA.Object).
- Pseudo-objets : mappés en classes Java.

Types primitifs Java et IDL

1. Types simples : correspondance directe en Java pour $double, char, boolean, float, short$. Autres : $octet$ en $byte$, $string$ en $String$, $long$ en int.
2. Types complexes = classes Java ($typeHelper$).

Les exceptions

```
exception a{};
exception b{};

interface c {
   void m() raises (a,b);
}
```

devient :

```
void m() throws a,b {
  ...
  throw new a();
}
```

$\Longrightarrow$ exception utilisateur = classe héritant de java.Exception (exception contrôlée).

Passage de paramètres

- Paramètres passés en *in* :

```
long write(in long p);
```

devient :

```
int write(int p);
```

- Paramètres passés en *inout* ou *out* : Java = passage par valeur, sauf pour les objets $\Longrightarrow$ utilisation de classes $Holder$.

```
long ioctl(inout long param);
```

devient :

```
int ioctl(IntHolder param);
```

Structures IDL

```
struct stat_data   {
  string device;
  long inode;
  long uid;
};
```

Une structure IDL est projetée en une classe Java:

```
public final class stat_data
 implements org.omg.CORBA.portable.IDLEntity {
  public java.lang.String device = "";
  public int inode;
  public stat_data(java.lang.String device, int inode, ...)
    this.device = device;
    ...
```

Séquence IDL

```
typedef sequence<stat_data> stat_table;
```

Une séquence IDL est un tableau Java:

```
stat_data [] s = new stat_data [3];

s[0] = new stat_data();
s[0].device="/dev/sd1";
s[0].inode=12134;
s[0].uid=110;

s[1] = new stat_data();
s[1].device="/dev/sd2";
s[1].inode=46545;
s[1].uid=100;

s[2] = new stat_data();
...
```

Enumération IDL

```
enum mode {read_only, write_only, read_write};
```

Une énumeration IDL est projetée en une classe Java:

```
public final class mode
 implements org.omg.CORBA.portable.IDLEntity {
  private int value = -1;
  public static final int _read_only = 0;
  public static final mode read_only = new mode(_read_only),
  public static final int _write_only = 1;
  public static final mode write_only = new mode(_write_only
  ..
  public int value() {
   return value;
}
  public static mode from_int(int value) ...
```

Serveur CORBA simplifié

```
ORB orb = ORB.init();
POA poa = POAHelper.narrow(orb.resolve_
  initial_references("RootPOA"));
poa.the_POAManager().activate();

compte_impl cpt = new compte_impl();
org.omg.CORBA.Object ocpt = poa.servant_to_reference(cpt);

String ior = orb.object_to_string(ocpt);
PrintWriter myfile = new PrintWriter(new FileOutputStream("myfile.txt"));
myfile.println(ior);
...
orb.run();
```

Invocation statique simplifiée en Java

```
ORB orb = ORB.init();

org.omg.CORBA.Object obj;
String ior = "IOR:0109C024F2353AE...";

obj = orb.string_to_object(ior);

compte cpt=compteHelper.narrow(obj);
if (cpt==null)
  System.err.println("Echec lors du narrow");

solde=cpt.debiter(1000);
```

Serveur CORBA avec Nommage

```
ORB orb = ORB.init();
POA poa = POAHelper.narrow(orb.resolve_
  initial_references("RootPOA"));
poa.the_POAManager().activate();

compte_impl cpt = new compte_impl();
org.omg.CORBA.Object ocpt = poa.servant_to_reference(cpt);

org.omg.CORBA.Object sn;
sn = orb.resolve_initial_references("NameService");

NamingContext nom = NamingContextHelper.narrow(sn);
if (nom==null)
  System.err.println("Echec lors du narrow");

nom.bind(ocpt,"nom du service");

orb.run();
```

Invocation statique avec Nommage

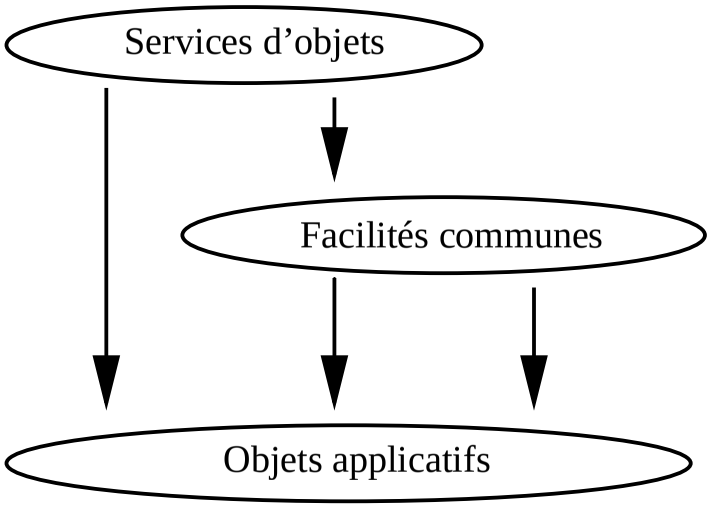
```
ORB orb = ORB.init();

org.omg.CORBA.Object obj;
org.omg.CORBA.Object sn;

sn = orb.resolve_initial_references("NameService");

NamingContext nom = NamingContextHelper.narrow(sn);
if (nom==null)
  System.err.println("Echec lors du narrow");

obj = nom.resolve("nom du service");

compte cpt=compteHelper.narrow(obj);
if (cpt==null)
  System.err.println("Echec lors du narrow");

solde=cpt.debiter(1000);
```

Sommaire

1. Concepts de base et architecture.
2. Le langage de description d'interface.
3. Zoom sur le bus à objets.
4. Mapping vers C++ et Java.
5. Les services et les facilités CORBA.
6. CORBA et l'interopérabilité.
7. Conclusion.
8. Références.

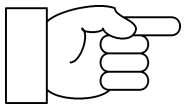
Les services et facilités CORBA (1)

- Les facilités communes[OMG 97a] et les services d'objets [OMG 97b] sont des composants logiciels. Objectif : alléger la réalisation d'applications CORBA.



- Limite mouvante entre les trois groupes de composants.
- Service/facilité = interfaces + implémentations.
- Raffinement possible (par héritage, par modification d'implémentations).

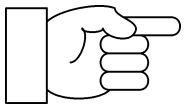
Les services et facilités CORBA (2)



Les facilités sont classées en deux groupes :

- Les facilités horizontales, qui sont des composants pouvant servir dans plusieurs domaines d'applications (ex : facilité d'impression).
- Les facilités verticales, qui sont spécifiques à un domaine = objets métiers (ex : paye, gestion commercial, gestion hospitalière).

Les services et facilités CORBA (3)



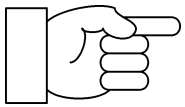
Pas de classement pour les services d'objets, qui sont des objets nécessaires à la construction des applications. Exemples de services normalisés à ce jour :

- Service de nom, d'événements, de cycle de vie.
- Service de persistence.
- Service de sécurité.
- Service transactionnel.
- Service de contrôle de la concurrence.
- Service de temps.
- etc.

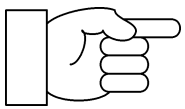
Exemples de services

1. Le service de noms.
2. Le service de cycle de vie.
3. Le service d’événements.

Le service de noms (1)



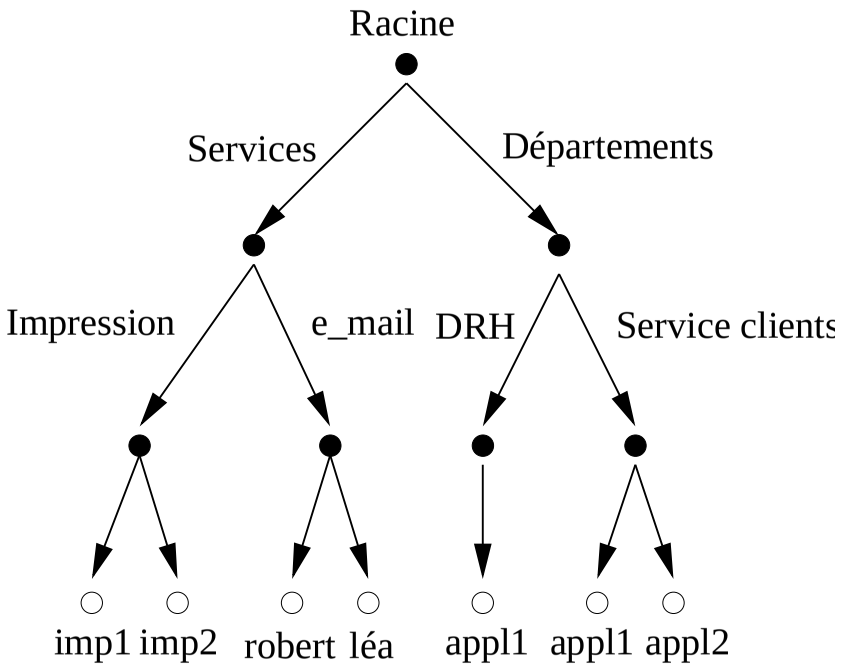
Permet aux serveurs d'associer un nom symbolique à un objet. Les clients utilisent alors ces noms symboliques pour obtenir des références sur les objets en question (mécanismes de liaison/résolution).



Ce service introduit les notions de :

- Nom symbolique. Un nom est unique dans un espace de nommage.
- Contexte, qui définit un espace de nommage. Les contextes peuvent être concaténés.

Le service de noms (2)



- Contextes + noms = arborescences.
- Feuilles = objets ; nœuds = contextes.
- Fonctionnement proche d’un système de fichiers hiérarchique.
- Notion de "Binding" (ou de "liaison").

Le service de noms (3)

- Binding ou liaison : association entre un nom symbolique et une référence d'objet.
- Binding de contexte versus Binding d'objet.

- **Structure de données :**
  - **NameComponent** : composant élémentaire d'un chemin d'accès à un contexte ou objet. Nom symbolique.
  - **Name** : nom symbolique complet.

Le service de noms (4)

```
module CosNaming {

  struct NameComponent {
    string id;
    string kind;
  }

  enum BindingType {nobject, ncontext};

  typedef sequence<NameComponent> Name;

  struct Binding {
    Name binding_name;
    BindingType binding_type;
  };

  typedef sequence<Binding> BindingList;
```

Le service de noms (5)

```
  interface BindingIterator {
    boolean next_one(out Binding b);
  };

  interface NamingContext {

  exception NotFound{};
  exception InvalidName{};
  exception AlreadyBound{};

    void bind(in Name n, in Object obj)
      raises(NotFound, AlreadyBound, InvalidName);
    NamingContext bind_new_context(in Name n);
    void unbind(in Name n);
    Object resolve(in Name n);
    void list(in long how_many, out BindingList bl,
      out BindingIterator bi);
  };
};
```

Le service de noms (6)

- Rappel Mapping Java

  - struct IDL => classe Java.
  - enum IDL => classe Java.
  - sequence IDL = > tableau d’objets.

Service : API à retenir

```
module CosNaming {

  struct NameComponent {
    string id;
    string kind;
  }
  typedef sequence<NameComponent> Name;

  interface NamingContext {
     void bind(in Name n, in Object obj)
        raises(NotFound, AlreadyBound, InvalidName);
     NamingContext bind_new_context(in Name n);
     Object resolve(in Name n);
  }
```

Exemple: liaison d’un contexte

```
Object Obj=orb.resolve_initial_references("NameService");
NamingContext root=NamingContextHelper.narrow(obj);

NameComponent [] n1 = new NameComponent [1];
n1[0] = new NameComponent();
n1[0].id = "services";
n1[0].kind = "";
NamingContext nc1 = root.bind_new_context(n1);
```

Exemple: liaison (relative) d'un objet

```
Object Obj=orb.resolve_initial_references("NameService");
NamingContext root=NamingContextHelper.narrow(obj);

NameComponent [] n1 = new NameComponent [1];
n1[0] = new NameComponent();
n1[0].id = "services";
n1[0].kind = "";
NamingContext nc1 = root.bind_new_context(n1);

n1[0].id = "impression";
NamingContext nc2 = nc1.bind_new_context(n1);

n1[0].id = "imp1";
nc2.bind(n1, PTR);
```

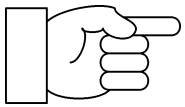
Exemple: liaison (absolue) d'un objet

```
Object Obj=orb.resolve_initial_references("NameService");
NamingContext root=NamingContextHelper.narrow(obj);

NameComponent [] n1 = new NameComponent [3];
n1[0] = new NameComponent();
n1[0].id = "services";
n1[0].kind = "";
n1[1] = new NameComponent();
n1[1].id = "impression";
n1[1].kind = "";
n1[2] = new NameComponent();
n1[2].kind = "";
n1[2].id = "imp1";
root.bind(n1, PTR);
```

Exemple: résolution (absolue) d'un objet

```
Object Obj=orb.resolve_initial_references("NameService");
NamingContext root=NamingContextHelper.narrow(obj);

NameComponent [] n1 = new NameComponent [3];
n1[0] = new NameComponent();
n1[0].id = "services";
n1[0].kind = "";
n1[1] = new NameComponent();
n1[1].id = "impression";
n1[1].kind = "";
n1[2] = new NameComponent();
n1[2].id = "imp1";
n1[2].kind = "";
Object PTR=root.resolve(n1);
```

Le service de cycle de vie (1)



Offrir des outils de gestion des objets CORBA.

- Opération de création :
  - Utilisation d'usines à objets spécifiques.

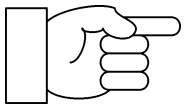
```
interface doc;
interface usine_docs {
  doc create_doc(in param ...);
};
```

  - Usines à objets génériques $\Longrightarrow$ politiques de gestion des ressources.
- Opérations de suppression, copie ou migration d'objets (ou de graphes d'objets).

Le service de cycle de vie (2)

```
module CosLifeCycle {
  interface FactoryFinder {
    Factories find_factories(in Key factory_key);
  };

  interface LifeCycleObject {
    LifeCycleObject copy(in FactoryFinder there,
      in Criteria c);
    void move(in FactoryFinder there, in Criteria c);
    void remove();
  };

  interface GenericFactory {
    Object create_object(in Key k, in Criteria c);
    ...
  };
};
```

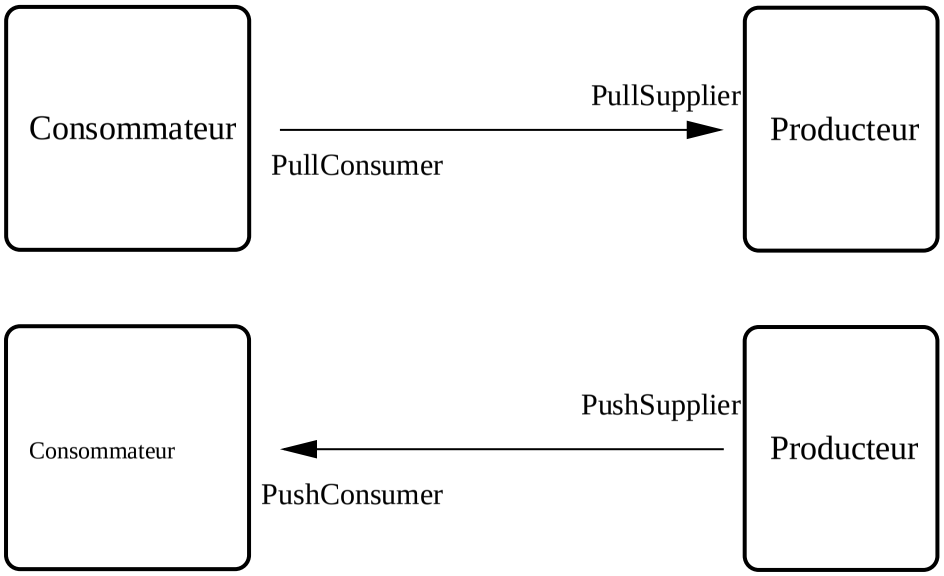
Le service d'événements (1)



CORBA offre un modèle d'invocation principalement synchrone $\Longrightarrow$ ce service doit permettre l'échange asynchrone d'événements de producteurs vers des consommateurs.

- Service très populaire par sa souplesse.
- La spécification laisse une large liberté quant aux différentes implémentations possibles.
- Combinaisons possibles :
  - Initiateur des communications.
  - Événements "génériques" ou typés.
  - Utilisation de canaux.

Le service d'événements (2)

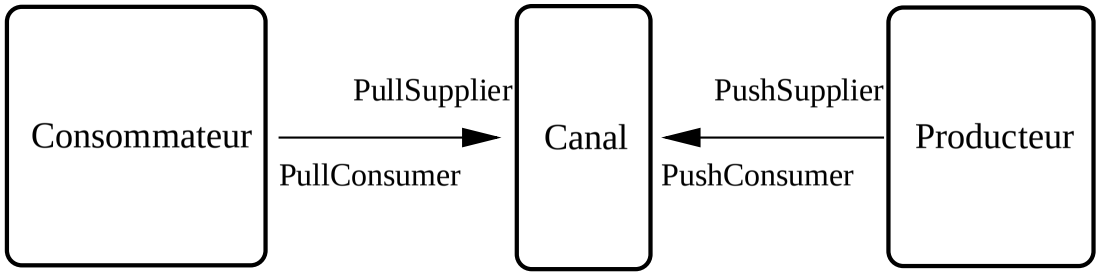


- Communications à l'initiateur du producteur (*push*) ou du consommateur (*pull*).
- Échange des références d'objets, identités connues.
- Ne permet pas le découplage producteur/consommateur.

Le service d'événements (3)

```
module CosEventComm {
  interface PushConsumer {
    void push(in any event);
    void disconnect_push_consumer();
  };
  interface PushSupplier {
    void disconnect_push_supplier();
  };
  interface PullSupplier{
    any pull();
    any try_pull(out boolean has_event);
    void disconnect_pull_supplier();
  };
  interface PullConsumer {
    void disconnect_pull_consumer();
  };
};
```

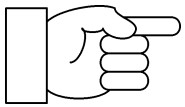
Le service d'événements (4)



- Canal = objet CORBA permettant de découpler les communications entre producteurs/consommateurs.
- Utilisation de filtres. Combinaisons de canaux. QoS non spécifiée.
- Identités inconnues.
- Interfaces permettant de créer, détruire des canaux, connecter des producteurs/consommateurs.
- Événements typés possibles.
- Communication n vers n possible.

Sommaire

CORBA et l'interopérabilité (1)

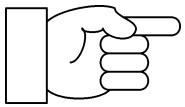


Objectif = étendre l'interopérabilité entre des ORB différents, voir des systèmes objets "non CORBA".

- Que faut-il pour interopérer ?

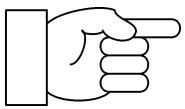
1. Manipuler un modèle objet identique.
2. Être capable d'interpréter convenablement une référence d'objet (ex : localiser son implémentation).
3. Être capable de véhiculer les invocations et les exceptions.
4. Manipuler une représentation communes des types, indépendamment des systèmes sous jacent (condition nécessaire dans un environnement constitué d'un seul ORB).

CORBA et l'interopérabilité (2)



Solution apportée dans CORBA 2 (CORBA 1 normalise uniquement l'interface de programmation). La version 2 définit :

- Des mécanismes de ponts.
- Un protocole standard : GIOP [a], ainsi que sa correspondance sur TCP/IP : IIOP [b].
- Des environnements spécifiques (ex : interopérabilité avec DCOM ou par DCE).



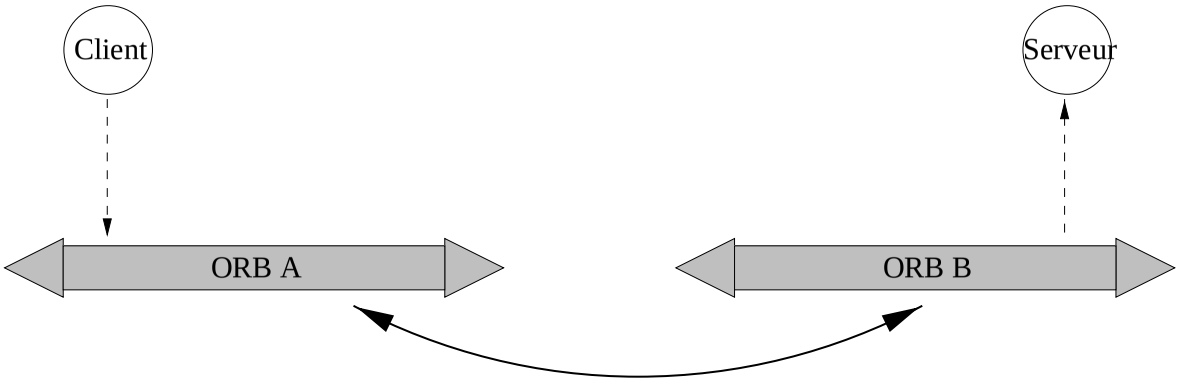
Compatible CORBA 2 = API CORBA + IIOP.

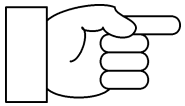
---

[a] **GIOP** pour **G**lobal **I**nter-**O**RB **P**rotocol.

[b] **IIOP** pour **I**nternet **I**nter-**O**RB **P**rotocol.

CORBA et l'interopérabilité (3)

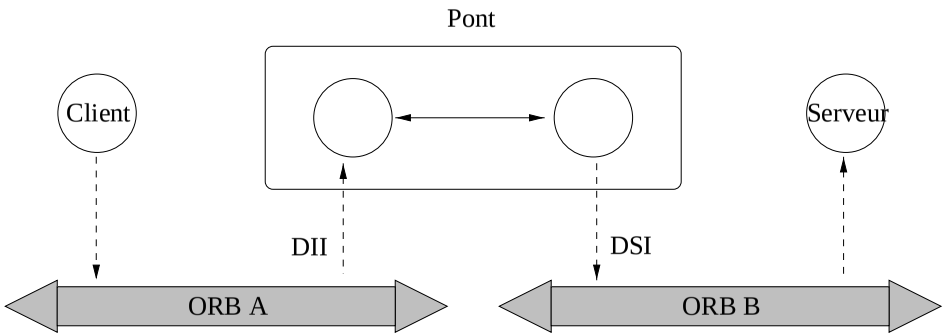




Utilisation native de IIOP.

- Interopérabilité d'office.
- Le plus courant dans les ORB commercialisés aujourd'hui.

CORBA et l'interopérabilité (4)



- Permet la mise en œuvre de protocoles optimisés, adaptés à certains types d'environnements ou d'applications.
- Conservation de l'existant.
- Le pont mémorise des objets *proxies*.
- Conversion de protocole, de référence d'objet, voir de modèle objet.

- Combinaison de ponts + IIOP en fond de panier.

Les références d'objet (1)

☞ Référence d'objet = type opaque pour l'utilisateur. Dans un même domaine de référence, les différents ORB doivent pouvoir interpréter les références d'objets $\Longrightarrow$ utilisation d'IOR [a] :

☞ Une référence d'objet c'est avant tout une information sur une extrémité, vis-à-vis d'un protocole (ex : adresse IP + port + clef d'objet).

[a] **IOR** pour **I**nteroperable **O**bject **R**eference.

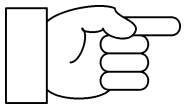
Les références d'objet (2)

```
module IOP {
  typedef unsigned long ProfiledId;
  struct TaggedProfile {
    ProfileId tag;
    sequence<octet> profile_data;

  struct IOR {
    string type_id;
    sequence<TaggedProfile> profiles;
  };
};
```

- L'OMG n'impose pas l'utilisation des IOR pour les communications internes d'un ORB.
- Un IOR peut il traverser plusieurs protocoles de transport

? En théorie oui, en pratique, ca reste peu utile !

Présentation de GIOP (1)

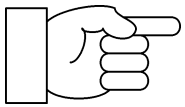


Protocole permettant la communication entre deux ORB CORBA potentionellement différents, afin de :

- Transmettre les invocations d'objets et les exceptions.
- Offrir un service de localisation pour implanter des outils de migration d'objets.
- Assurer le support d'opérations sur des références d'objet (*Object::get_interface*(), *Object::get_implementation*(), *Object::is_a*(), et *Object::non_existent*()).

$\Longrightarrow$ Support de GIOP en "natif" ou sous forme de pont.

Présentation de GIOP (2)



Caractéristiques :

- Protocole asymétrique ; notion de client et de serveur (ex : un ORB ou un pont).
- Protocole orienté connexion. Le client est l'initiateur des connexions. Le serveur ne peut pas en ouvrir. Multiplexage autorisé.
- Emballage CDR [a] des messages.
- Nécessite l'utilisation d'une couche transport fiable. Gestion des désequencements de messages non nécessaire.
- Sept types de messages seulement car protocole minimal = surcoût minimal.

[a] **CDR** pour **C**ommon **D**ata **R**epresentation.

Les messages GIOP (1)

- Messages constitués de trois parties :
  - Entête GIOP.
  - Entête spécifique au type de message.
  - Corps de message optionnel.

```
module GIOP {
  struct MessageHeader {
    char magic[4];
    Version GIOP_version;
    boolean byte_order;
    octet message_type;
    unsigned long message_size;
  };
```

Les messages GIOP (2)

- Émission d'une invocation.

```
module GIOP {
  struct RequestHeader {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    boolean response_expected;
    sequence<octet> object_key;
    string operation;
    Principal requesting_principal;
  };
};
```

- Le corps optionnel est constitué d'une structure comportant les paramètres *in* et *inout*.

Les messages GIOP (3)

- Transfert des résultats chez le client.

```
module GIOP {
  enum ReplyStatusType {
    NO_EXCEPTION, LOCATION_FORWARD,
    USER_EXCEPTION, SYSTEM_EXCEPTION
  };
  struct ReplyHeader {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    ReplyStatusType reply_status;
  };
};
```

- Le corps optionnel de la réponse (exécution réussie, exception levée ou objet absent).
- LOCATION_FORWARD transparent pour l'application. Réémission par le client GIOP.

Les messages GIOP (4)

- Gestion de la localisation d'objet :

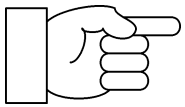
```
module GIOP {
  enum LocateStatusType {
    UNKNOWN_OBJECT, OBJECT_HERE, OBJECT_FORWARD
  };
  struct LocateRequestHeader {
    unsigned long request_id;
    sequence<octet> object_key;
  };
  struct LocateReplyHeader {
    unsigned long request_id;
    LocateStatusType locate_status;
  };
};
```

- Fonctionnalité transparente à l'application. Corps optionnel composé d'un IOR.

Les messages GIOP (5)

- Autres messages :
  - Annulation par le client d’une requête (CancelRequest).
  - Fermeture d’une connexion (CloseConnection).
  - Erreur de protocole (MessageError).

Le protocole IIOP



Implantation de GIOP sur TCP/IP (TCP).

- Description de profil GIOP pour IIOP (valeur du champs tag = TAG_INTERNET_IOP) :

```
module IIOP{
  struct Version {
    char major;
    char minor;
  };

  struct ProfileBody {
    Version iiop_version;
    string host;
    unsigned short port;
    sequence<octet> object_key;
  };
};
```

Sommaire

Synthèse, conclusion (1)

1. Modèle d'objets répartis :
   - Transparence à la localisation, intégration, portabilité, interopérabilité.
   - Séparation interface et implémentation. Référence. IDL et Mapping.
   - Mécanisme synchrone. Sémantique exactement une fois ou au plus une fois.
2. Interopérabilité :
   - Les protocoles GIOP et IIOP.
   - Notion de référence interopérable d'objet.
   - Format de données (CDR).
   - Considérations architecturales.
3. Exemples de services.

Synthèse, conclusion (2)

- Performance $\Longrightarrow$ coût de la généralité.
- Modèle structurant (modèle objet).
- Facilite le développement (1 ligne d'IDL = 50 lignes de C++/Java).
- Applications réparties ? $\Longrightarrow$ état global, sémantiques/mécanismes.
- Autres approches : OLE/DCOM, Java RMI , .NET ...

Sommaire

Références

[OMG 93] OMG. « The Common Object Request Broker Architecture and Specification, Revision 1.2 ». TC Document 93-12-43, December 1993.

[OMG 97a] OMG. « Common Facilities Architecture ». TC Document 97-06-15, November 1997.

[OMG 97b] OMG. « CORBAservices : Common Object Services Specification ». TC Document 97-12-02, November 1997.

[OMG 99] OMG. « The Common Object Request Broker : Architecture and Specification. Revision 2.3 ». TC Document 99-10-07, October 1999.