Réalisation d'un sous-système Saturne tolérant les pannes sur CHORUS/COOL

Rapport de DEA Systèmes Informatiques
Année 1995/96

Singhoff Frank

23 septembre 1996

Résumé

Devant la croissante complexité des systèmes embarqués, les concepteurs et réalisateurs de logiciels ont besoin d'outils adaptés. Ces outils doivent leur permettre de spécifier, réaliser et certifier efficacement leurs applications embarquées. En dehors de leur réalisation, l'exécution de ces logiciels est également très difficile à maîtriser car ils ont de très fortes contraintes temps réel et de sûreté de fonctionnement. Parallèlement, il devient aujourd'hui de plus en plus nécessaire d'utiliser des systèmes distribués, et ce pour des problèmes de performances, de localisations des équipements physiques (tel que des capteurs), de tolérance aux pannes ou à des fins de réduction de coût du système. Malheureusement, l'exécution d'applications temps réel dans un environnement distribué reste encore mal dominée. Sur cette architecture distribuée, il est nécessaire de construire un environnement d'exécution permettant au logiciel applicatif de respecter ses contraintes. L'environnement d'exécution devra en particulier garantir le déterminisme des temps d'exécution et de communication. Saturne est un modèle d'exécution pour les applications temps réel embarquées dans un environnement distribué élaboré par le **C**entre d'**E**tudes et de **R**echerches de **T**oulouse. Ce document propose des mécanismes de tolérance aux pannes utilisables dans une implantation du modèle Saturne avec un système d'exploitation distribué temps réel : le système CHORUS.

Mots clefs

Temps réel à contraintes critiques, avionique modulaire, Saturne, Esterel, CHORUS, COOL, tolérance aux pannes, systèmes distribués, CORBA

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé à réaliser ce travail. Je remercie tout particulièrement Eric Gressier, Maître de Conférence au Conservatoire National des Arts et Métiers pour son suivi et ses conseils durant ce travail. Je le remercie aussi pour son soutien durant ces trois dernières années d'études au Conservatoire National des Arts et Métiers. Je remercie aussi tout le personnel du département études logiciels de Dassault Aviation qui m'ont apporté leur aide et spécialement Claire Campan, ingénieur, qui reçoit toute ma gratitude et ma sympathie pour ses conseils permanents, pour m'avoir fourni l'aide technique lors de la phase de développement et pour son encadrement. Je remercie aussi Christine Ledey pour ses patientes explications et son aide à l'intégration de l'application de suivi de terrain dans le deuxième prototype ainsi que Yann Le Biannic pour son soutien et toutes les démarches qu'il a entreprises pour que cette étude s'effectue dans de bonnes conditions. Enfin, je salue toutes les autres personnes, du Conservatoire National des Arts et métiers, de Dassault Aviation et d'ailleurs qui m'ont offert leur soutien durant ces quatre dernières années.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Aujourd'hui, la société Dassault Aviation spécifie les systèmes de ses avions et ce sont des équipementiers (Thomson, Sagem, Sextant Avionique, etc) qui réalisent ces équipements. Le choix des composants matériels et logiciels est propre à chaque équipementier. Cette hétérogénéité entraîne un surcoût de fabrication et de maintenance.

L'avionique modulaire a pour objectif de réduire ce surcoût en définissant une architecture matérielle et logicielle permettant l'utilisation de logiciels dits "sur étagère". Par logiciels "sur étagère", on entend des composants logiciels standards pouvant être achetés dans l'industrie. On évoque souvent ce concept par le terme de COTS[1]. L'utilisation de logiciels sur étagère dans les systèmes embarqués fait actuellement l'objet d'une étude par un groupement d'industriels européens et américains : l' OSAF[2][45]. L'avionique modulaire doit aussi permettre de réduire le coût d'une mission en augmentant le nombre d'heures de vol sans maintenance. Pour cela, les systèmes des avions devront intégrer des mécanismes de tolérance aux pannes pour compenser d'éventuelles défaillances en vol.

Parallèlement à l'avionique modulaire, Dassault Aviation étudie des méthodes de spécification de logiciels temps réel embarqués où les applications s'exécutent selon un modèle synchrone faible[8] élaboré par le CERT[3] : le modèle Saturne[17, 1, 50]. CHORUS[65] étant le système d'exploitation évalué dans le cadre des études d'avionique modulaire, il est également choisi pour cette étude dont l'objectif est de déterminer dans quelles conditions le

[1]COTS pour **C**ommercial **O**n **T**he **S**helf products.

[2]OSAF pour **O**MI **S**oftware **A**rchitecture **F**orum. Ce groupement comprend entre autres Chorus Systèmes, IONA Technologies, l'Object Management Group, Interglossa, etc.

[3]CERT pour **C**entre d'**E**tudes et de **R**echerches de **T**oulouse.

modèle Saturne peut être mis en oeuvre sur un système d'exploitation distribué temps-réel. Une étude des mécanismes de tolérance aux pannes ainsi que l'analyse des apports des technologies CORBA[4][44, 72] dans le développement d'applications temps réel complètent ces objectifs. C'est pourquoi nous avons utilisé le bus à objets COOL[5][66] pour la mise en oeuvre de nos différents prototypes.

Ce document traite de l'étude des mécanismes de tolérance aux pannes que nous avons appliqués sur Saturne. Le modèle Saturne qui existe depuis 1992 a donné lieu à de nombreuses études. On peut citer à titre d'exemples les extensions multi-synchrones[51] de Saturne proposées par le CERT en collaboration avec Dassault Aviation, ou le modèle Saturne objet[22] conçu par Y. Faure. Mais à notre connaissance, l'étude des mécanismes de tolérance aux pannes sur Saturne ne semble pas être un problème qui a été abordé. Dans un premier temps, nous présenterons dans le chapitre deux le modèle synchrone fort, puis nous définirons dans le chapitre trois le modèle Saturne. Le chapitre quatre sera consacré aux mécanismes de tolérance aux pannes dans les systèmes distribués. La tolérance aux pannes dans les systèmes distribués est un domaine vaste, aussi nous présenterons les mécanismes habituellement utilisés à travers quelques exemples de systèmes intégrant les contraintes qui nous sont posées : l'utilisation d'un système à objets dans un environnement temps réel à contraintes critiques. Enfin, avant de conclure dans le chapitre six, nous détaillerons dans le chapitre cinq le sous-système Saturne sur CHORUS/COOL ainsi que ses mécanismes de tolérance aux pannes. Nous renvoyons le lecteur à [21] pour de plus amples informations sur l'environnement d'exécution (CHORUS, CORBA et COOL) ainsi que sur les conclusions concernant l'adéquation de CHORUS et de COOL à la mise en oeuvre de Saturne.

[4]CORBA pour **C**ommon **O**bject **R**equest **B**roker **A**rchitecture.
[5]COOL pour **C**HORUS **O**bject **O**riented **L**ayer.

Chapitre 2

Le modèle synchrone fort

Dans ce chapitre, nous définirons le modèle synchrone fort. Puis, nous décrirons le langage synchrone utilisé dans Saturne: Esterel. Nous présenterons finalement les preuves qui peuvent être effectuées sur les programmes écrits en Esterel.

2.1 Introduction aux langages synchrones

*** Notion de systèmes réactifs**

La notion de systèmes réactifs fut introduite par D. Harel et A. Pnueli[48]. Par systèmes réactifs, on entend généralement un système qui réagit immédiatement à des entrées en provenance d'un environnement extérieur en fournissant des sorties instantanément. Par exemple, on peut citer les processus de contrôle industriel en temps réel, les automatismes (distributeurs de boissons, billets de banque), etc. Ces systèmes réactifs ont été opposés aux systèmes transformationnels par A. Pnueli. Les systèmes transformationnels possèdent leurs entrées lors du démarrage de leur exécution, et fournissent un résultat à la fin de leur exécution. On considère souvent un troisième type de système qui est le système interactif. Ce dernier réagit aussi à des entrées événementielles, mais à son rythme.

*** Le modèle synchrone fort**

Dans le domaine d'application des systèmes réactifs, les langages procéduraux ne sont pas adaptés. Généralement, pour développer ce type de système, on utilisait plutôt :

- Des langages asynchrones comme CSP[1]/OCCAM ou ADA : malheureusement, le type de parallélisme introduit dans ces langages apporte aussi de l'indéterminisme, ce qui est à proscrire dans les systèmes temps réel critiques comme les systèmes embarqués,

- Des automates d'états finis : ceux-ci deviennent très vite compliqués à concevoir quand leur taille est importante,

- Des primitives système de bas niveau : elles ne permettent pas d'effectuer des certifications de logiciel.

De plus, toutes ces méthodes ne permettent pas d'exprimer les contraintes de temps d'exécution (bien qu'il existe parfois des extensions comme c'est le cas pour CSP[13]). Pour développer ce type de système, des langages spécifiques ont donc été proposés : les langages synchrones. Les langages synchrones les plus connus sont Lustre[46], Signal[27, 39] et le plus ancien des trois : Esterel[4, 6, 7]. L'avantage essentiel de ces langages est qu'ils font cohabiter, grâce au modèle synchrone fort, à la fois le parallélisme d'expression, et une exécution déterministe. De plus, ils permettent de mettre en oeuvre des preuves logiques et temporelles des applications.

Le modèle synchrone fort repose sur un certain nombre de concepts :

- L'hypothèse fondamentale du synchronisme : ce modèle suppose que les temps de réaction des noyaux[2] doivent être nuls. En d'autres termes, on suppose dans ces langages que la vitesse de calcul des machines est infiniment rapide. Cette hypothèse qui semble absurde dans la réalité est en fait tout à fait réaliste. En effet, dans ce type de langage on ne considère pas le temps physique mais plutôt "l'instant d'activation", c'est à dire le moment où les signaux d'entrées arrivent au système réactif. **Le temps physique est discrétisé et on utilise un temps logique**. L'intérêt essentiel de cette hypothèse est qu'elle simplifie les preuves de programmes. Les contraintes temporelles sont alors garanties par l'environnement d'exécution du programme synchrone qui doit s'assurer qu'à l'instant t, les traitements de l'instant t-1 sont terminés. Dans la réalité, pour valider l'hypothèse du modèle synchrone fort, il faut que l'intervalle entre deux instants d'activation soit supérieur au temps de calcul du système réactif : un jeu de signaux ne doit pas arriver au système avant que le jeu précédent ne soit complètement traité.

[1] CSP pour **C**ommunicating **S**equential **P**rocesses.

[2] Par noyau, nous entendons système réactif.

En ne mélangeant pas les différents jeux de signaux, on fournit aux systèmes réactifs une propriété importante qui est nécessaire aux systèmes temps réels : leur déterminisme. Un programme déterministe est un programme qui délivre toujours les mêmes sorties quand il reçoit les mêmes entrées.

- Diffusion des signaux : Le modèle synchrone fort offre une diffusion **instantanée** des données entre les différents modules.

- Atomicité de l'exécution du système réactif : quand le système réactif reçoit ses données d'entrée, il s'exécute instantanément et surtout de manière **atomique**.

2.2 Description du langage Esterel

2.2.1 Syntaxe et sémantique du langage Esterel

*** Présentation d'Esterel**

Esterel[6, 5] fut conçu par G. Berry et L. Cosserat. Contrairement à Lustre et Signal, Esterel est un langage impératif. Il permet comme les deux langages précédents une gestion aisée de la concurrence. En effet, il contient les constructions nécessaires pour exprimer l'exécution de tâches en parallèle. Le parallélisme d'Esterel est un parallélisme d'expression. Le code généré est un code séquentiel où les actions parallèles sont sérialisées. Cette caractéristique est un avantage dans les environnements temps réel car **il n'y a aucun indéterminisme quant à l'ordonnancement des tâches Esterel, ce qui simplifie la mise au point des programmes.** Ce parallélisme d'expression est un point commun avec Lustre et Signal. Comme les langages synchrones précédents, Esterel peut faire l'objet de preuves formelles car sa compilation permet d'obtenir des automates d'états finis.

Dans ce paragraphe, nous verrons les instructions impératives du langage, puis nous décrirons comment les modules Esterel communiquent entre eux. Enfin nous regarderons comment sont gérés le temps et les tâches dans Esterel.

*** Les instructions impératives**

Une application Esterel est découpée en modules. Chaque module est constitué d'une partie déclarative (l'interface du module) et d'une partie où

Instructions	Commentaires
nothing	ne fait rien (ne prend aucun temps d'exécution)
a;b	composition séquentielle : a et b sont exécutés séquentiellement et dans cet ordre
halt	prend un temps infini : la tâche ne sort jamais de cette instruction
a ‖ b	exécute l'instruction (ou la tâche) a en parallèle avec b
var:=expression	instruction d'affectation
[*a*‖*b*];*c*	les caractères [et] permettent d'imposer des priorités entre les opérateurs et les instructions : ici on exécute a et b en parallèle, puis quand a et b sont terminés, on exécute c.
loop instructions *endloop*	effectue une boucle infinie sur la suite d'instructions
if exp *then* instructions1 *else* instructions2	test conditionnel classique
call p	appel de la procédure p

Tableau 2.1: Instructions impératives d'Esterel

sont décrites les instructions impératives et temporelles (implantation du module).

*** La partie déclarative d'un programme Esterel**

```
var foo in
        suite d'instructions;
end var;
```

Figure 2.1: Déclaration d'une variable en Esterel

Cette partie déclare :

- Les signaux d'entrée et de sortie du module qui sont décrits dans le paragraphe suivant,

- Les prototypes des tâches du module,
- Les prototypes des procédures et des fonctions,
- Les signaux de fin de tâche,
- Les relations,
- Les types construits par l'utilisateur,
- Les constantes.

REMARQUE :
En Esterel, la déclaration des variables est toujours locale à une suite d'instructions [3](voir exemple de la figure 2.1).

*** Liste des instructions impératives d'un programme Esterel**

Esterel possède des instructions de base telles que l'affectation, et des instructions de contrôle de flots (bien qu'elles soient moins nombreuses que pour des langages structurés comme Pascal ou C). Les principales instructions sont décrites dans le tableau 2.1.

```
loop
        x:=x+1:
end loop;
```

Figure 2.2: Boucle Esterel illégale

Dans les langages synchrones, un programme ne doit pas "prendre de temps" pour s'exécuter. En d'autres termes, toute instruction doit avoir un temps d'exécution fini . Une boucle décrite comme dans la figure 2.2 est illégale : en effet, cette boucle est une boucle infinie qui ne peut donc pas se terminer pendant une transition de l'automate et sera rejetée lors de la compilation. Une seule instruction peut transgresser cette règle : c'est l'instruction "*halt*" qui prend un temps infini et qui est utilisée pour stopper une application.

*** Les signaux et les capteurs Esterel**

[3]Ce mécanisme est identique aux déclarations de variables locales dans CAML[33].

Déclarations	Commentaires
input foo(*integer*);	signal en entrée du module véhiculant un entier
input foo;	signal en entrée sans donnée (dit signal pur)
output foo(COMPLEX);	signal en sortie du module transportant une donnée de type défini par l'utilisateur
inoutput foo;	signal en entrée et en sortie
signal foo *in* suite d'instructions; end;	signal local
return foo;	le signal foo est envoyé au module quand la tâche associée à ce signal est terminée
sensor foo(*integer*);	déclaration d'un capteur qui permet la lecture d'un entier

Tableau 2.2: Exemples de déclarations de signaux en Esterel

Comme nous l'avons précisé dans notre présentation, les modules Esterel communiquent entre eux par des signaux. Ces signaux leurs permettent aussi de communiquer avec l'environnement extérieur, le système d'exploitation par exemple. Un signal est à la fois un outil de synchronisation et un outil de communication. L'arrivée d'un signal "débloque" l'automate et démarre l'exécution de celui-ci. Un signal peut transporter optionnellement une information. Cette information peut être soit d'un type fourni par Esterel, soit d'un type construit par l'utilisateur avec le langage hôte. Esterel ne possède que peu de types : *integer, string* et *boolean*. Les types construits sont donc souvent utilisés. Chaque signal reçu de l'extérieur est diffusé à toutes les tâches dans le module, toutefois, un signal peut être déclaré localement à une partie du code (voir le tableau 2.2) : il est alors émis et reçu à l'intérieur du module.

Il existe plusieurs types de signaux :

- Les signaux en entrée,
- Les signaux en sortie,

- Les signaux en entrée et en sortie,
- Les signaux locaux,
- Les signaux locaux qui permettent de détecter la fin d'exécution d'une tâche,
- Et enfin les capteurs qui sont des signaux particuliers. En effet, on peut consulter leur valeur à n'importe quel moment et ils ne permettent pas de synchronisations.

```
emit foo(100);
```

Figure 2.3: Emission sur le signal foo de la valeur 100

```
case
        await SIG1 do instructions:
        await SIG2 do instructions;
        await ...
        await SIGn do instructions;
end case;
```

Figure 2.4: Réception de signaux grâce à une instruction *case/await*

```
wait foo;
```

Figure 2.5: Réception sur le signal foo

present S *then* instructions1 *else* instructions2;

Figure 2.6: Test de présence d'un signal

Des exemples de déclaration de tous ces signaux peuvent être consultés dans le tableau 2.2. On peut émettre un signal grâce à l'instruction "*emit*" (voir l'exemple 2.3), et recevoir grâce à "*await*". La réception de signaux peut prendre deux formes, une forme simple avec un seul "*await*", une forme plus complexe avec l'instruction "*case*". Le "*case*" possède une sémantique identique au PRI ALT d'OCCAM[34] (un exemple de "*case*" est donné dans la figure 2.4, un exemple de "*wait*" dans la figure 2.5). Enfin, Esterel permet de tester la présence d'un signal donné grâce à l'instruction "*present*" et d'exécuter des instructions selon le résultat du test (voir l'exemple 2.6).

*** Les instructions temporelles et les tâches**

*** La gestion des tâches**

Pour les communications comme pour la gestion des tâches, Esterel fait abstraction des services que peuvent offrir les couches inférieures (notamment, Esterel ne fait aucune supposition sur les services offerts par le système d'exploitation). Ainsi, quand un développeur veut utiliser une tâche Esterel, il doit réaliser une partie du développement dans le langage hôte. C'est en particulier le cas des fonctions qui permettent de créer, suspendre, réactiver et supprimer une tâche, mais c'est aussi le cas du corps des tâches. L'écriture du corps de la tâche dans le langage hôte permet de faire fonctionner une application avec des tâches écrites dans plusieurs langages différents. Un module central en Esterel prouvé effectue les changements de mode de fonctionnement de l'application, et un ensemble de tâches effectue des calculs de manière asynchrone par rapport au noyau Esterel (voir le chapitre trois traitant du modèle Saturne). Cette approche permet d'obtenir une application dont la partie critique (qui est la partie temps réel dur) est prouvée, et une partie moins critique non prouvée formellement.

Au niveau déclaration, il suffit de donner le prototype de la tâche. Comme pour les procédures, les tâches reçoivent des paramètres en entrée et en sortie. Il faut spécifier le type des paramètres dans deux ensembles entre parenthèses. Le premier ensemble cite les informations qui seront recopiées au retour de la tâche, le deuxième ensemble, les informations d'appel de la tâche. L'appel

```
task tacheUne (integer) (integer, integer, string)
        % cette tâche envoie comme réponse un entier et a besoin
        % de deux entiers et d'une chaîne comme paramètres d'appel
task tacheDeux () (string, string)
        % cette tâche ne renvoie pas de résultat mais
        % a besoin de deux chaînes pour commencer à s'exécuter
```

Figure 2.7: Déclaration de tâches Esterel

d'une tâche est faite par l'instruction "*exec*" (voir l'exemple de déclaration figure 2.7).

*** Les instructions temporelles**

```
do
        inst;
watching SIG1;
```

Figure 2.8: L'instruction *watching*

```
do
        inst;
watching SIG1;
timeout inst2;
```

Figure 2.9: L'instruction *watching* avec une clause *timeout*

```
every 60 SECONDE do
    emit MINUTE;
end every
```

Figure 2.10: L'instruction *every*

```
do
    inst;
    halt;
watching SIG
```

Figure 2.11: L'instruction *upto* simulée par un *watching*

```
do
    inst;
upto SIG
```

Figure 2.12: L'instruction *upto*

Les instructions temporelles d'Esterel sont un des points les plus importants du langage. Elles permettent d'implanter facilement dans les applications des mécanismes de chien de garde et d'exception. Le nombre de ces instructions et leurs utilisations étant très nombreux, nous n'en décrirons que quelques unes. Nous renvoyons le lecteur à [14] pour une description plus détaillée.

Le "*watching*" permet de mettre en place des mécanismes de type chien de garde : dans notre exemple 2.8, l'instruction "inst" est exécutée tant que le signal SIG1 n'est pas arrivé. Si l'instruction "inst" finit avant l'occurrence de SIG1 ou si le signal SIG1 arrive avant la fin de "inst", alors le programme sort du "*watching*". Le mot clef "*timeout*" permet d'exécuter une instruction si SIG1 arrive avant la fin de "inst". Dans notre exemple 2.9, si l'instruction "inst" n'est pas terminée lors de l'arrivée du signal SIG1, alors on exécute l'instruction "inst2".

L'instruction "*every*" permet d'exécuter un bloc d'instructions à chaque réception d'un signal. Dans notre exemple 2.10, à chaque réception de soixante signaux SECONDE, on envoie un signal MINUTE.

L'instruction "*upto*" est similaire à l'instruction "*watching*" mais le bloc instructions ne se termine pas : on pourrait simuler une instruction "*upto*" par l'exemple 2.11. La syntaxe de l'instruction "*upto*" est décrite dans l'exemple 2.12.

2.2.2 Exécution des programmes Esterel

Jusqu'à présent, nous avons décrit les principes d'Esterel, puis la syntaxe et la sémantique de ses instructions les plus simples (pour obtenir une définition de la sémantique d'Esterel plus formelle, on peut consulter [4, 7]) . Mais nous n'avons pas parlé de la manière dont est exécuté un code Esterel.

Comme nous l'avons déjà dit auparavant, Esterel fait une totale abstraction du système d'exploitation sur lequel il va fonctionner. Dans la pratique, une partie du code de l'application devra être écrite au moment de l'implantation. Le compilateur Esterel actuel génère un source (en C, ADA ou en LISP) constituant l'automate Esterel ainsi que les prototypes des fonctions, des procédures et des fonctions d'entrée/sortie des signaux de l'automate. Le programmeur est chargé de compléter le corps de ces fonctions.

Le code qui reste à développer concerne donc :

- La définition des types construits par l'utilisateur et les opérations sur ces types (voir l'exemple de l'additionneur de matrices à la fin de ce chapitre),
- Les primitives utilisées pour créer, suspendre, supprimer des tâches Esterel,
- L'implantation des fonctions, des procédures et des tâches Esterel.
- La communication entre automates. Esterel ne prend pas en compte les techniques qui sont utilisées par deux automates pour s'échanger les signaux, et laisse le choix au programmeur (en utilisant des sockets BSD, l'interface TLI[4], des appels de procédure à distance, des appels de méthode à travers un bus objet, etc),

[4]TLI pour **T**ransport **L**evel **I**nterface.

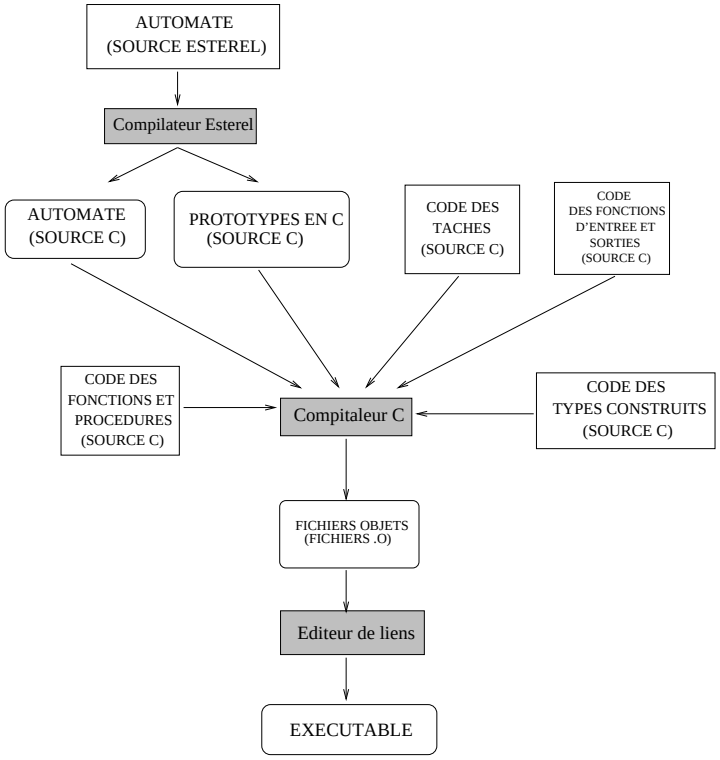

Figure 2.13: Construction d'un exécutable Esterel

Pour interfacer la partie générée et la partie à écrire dans le langage hôte, le compilateur Esterel attribue le nom des fonctions avec les règles suivantes : quand un automate de nom EMETTEUR[5] veut émettre un signal de nom INFOS, l'automate Esterel appellera la fonction C correspondante EMETTEUR_O_INFOS(donnée à envoyer). Le développeur devra donc écrire cette fonction, puis effectuer l'édition des liens avec le source généré par le compilateur Esterel. De la même manière, l'arrivée des signaux est constituée par une fonction C. Si l'automate reçoit un signal INF, l'utilisateur devra activer la fonction EMETTEUR_I_INF() pour signaler l'arrivée de la donnée. Un exemple de production d'un exécutable à base d'un automate Esterel est donné dans la figure 2.13.

Dans le source C que génère le compilateur Esterel, il existe une fonction[6] qui démarre l'exécution d'une transition de l'automate. Ainsi, quand l'uti-

[5]Le nom de l'automate est donné après le mot clef "*module*", voir exemple dans la partie suivante.

[6]Cette fonction a le nom du module Esterel.

lisateur désire activer un automate, il doit d'abord appeler les fonctions qui permettent de signaler l'arrivée des données, puis appeler cette fonction d'activation. Les fonctions d'émission de signaux sont automatiquement exécutées suivant les besoins de l'automate durant la réalisation de sa transition.

Il existe des outils qui permettent de simuler l'exécution de l'automate sans écrire la partie communication (comme l'outil XES[41]). Ces simulations obligent toutefois l'utilisateur à écrire le code concernant les types qu'il a construit et qui sont manipulés par l'automate. Ainsi, les développeurs peuvent tester le code Esterel sans se soucier des problèmes de communication.

2.2.3 L'exemple de l'émetteur-récepteur

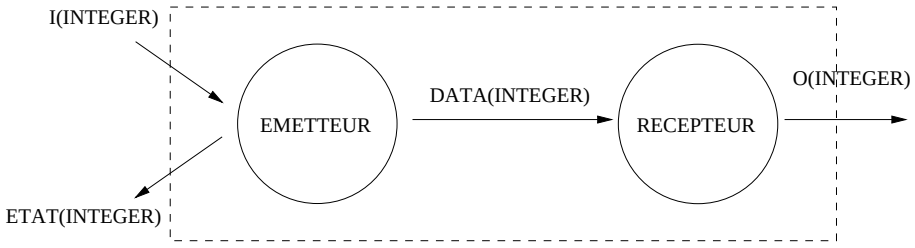

Figure 2.14: Exemple de l'émetteur et du récepteur

Dans cette partie et la suivante, nous allons présenter deux exemples de programmes Esterel qui constituent des "cas d'école" mais qui permettent d'illustrer les outils qui ont été utilisés dans cette étude. Ce premier exemple met en oeuvre deux modules : le module émetteur et le module récepteur. Ces modules manipulent des signaux transportant des entiers. Rappelons que les signaux Esterel peuvent véhiculer n'importe quel type d'information (les types offerts par Esterel ou définis par le programmeur) mais peuvent aussi ne transporter aucune information. Dans ce cas, ces signaux réalisent uniquement une synchronisation, ils sont alors appelés "signaux purs". La figure 2.14 nous montre la représentation graphique que nous adopterons dans ce document pour les applications Esterel. Chaque cercle constitue un module. Les arcs orientés définissent les signaux échangés. Le nom du signal est stipulé sur l'arc correspondant. Si le signal véhicule une information, alors son type est donné entre parenthèses après son nom. Le cadre en pointillé permet de délimiter l'application synchrone (qui est constituée de tous les modules Esterel) de l'environnement extérieur.

```
module emetteur:

input I (integer);
output ETAT (integer);
output DATA (integer);

every I do emit DATA(?I);
end every
||
loop
        await I;
        emit ETAT(1);
        await I;
        emit ETAT(2);
end loop
end module
```

Figure 2.15: Source Esterel de l'émetteur

*** Le module émetteur**

A chaque activation, ce module reçoit un entier par le signal I. Il réémet cette valeur sur le signal de sortie DATA en direction du module récepteur. La construction syntaxique exprimant qu'à chaque activation, le signal DATA doit être réémis est l'instruction "*every...do*". L'émission du signal ETAT est gérée différemment : l'instruction "*await*" suspend l'automate jusqu'à la prochaine transition. Ainsi, dans le module émetteur, la première transition émet un "1" sur le signal ETAT, puis un "2" sur la transition suivante et ainsi de suite. L'instruction "*loop*" exprime que cette suite d'émission des valeurs 1 et 2 sur le signal ETAT est faite dans une boucle infinie. Le symbole "?I" permet la lecture de la valeur reçue sur le signal I. La figure 2.15 donne le source Esterel de ce module. On remarquera que ce module est constitué de deux entités s'exécutant en parallèle. C'est le signe || qui exprime que les deux instructions *loop* et *every..do* sont parallèles.

*** Le module récepteur**

Le module récepteur lit un signal d'entrée DATA à chaque transition, puis, émet le signal O. La valeur émise sur O est la valeur reçue sur le signal d'entrée DATA plus la valeur du signal DATA à la transition précédente. La

```
module recepteur:

input DATA(integer);
output O (integer);

var LAST := 0 : integer in
every DATA do
      emit O (?DATA+LAST);
      LAST := ?DATA;
end every
end var
end module
```

Figure 2.16: Source Esterel du récepteur

figure 2.16 donne le source Esterel de ce module.

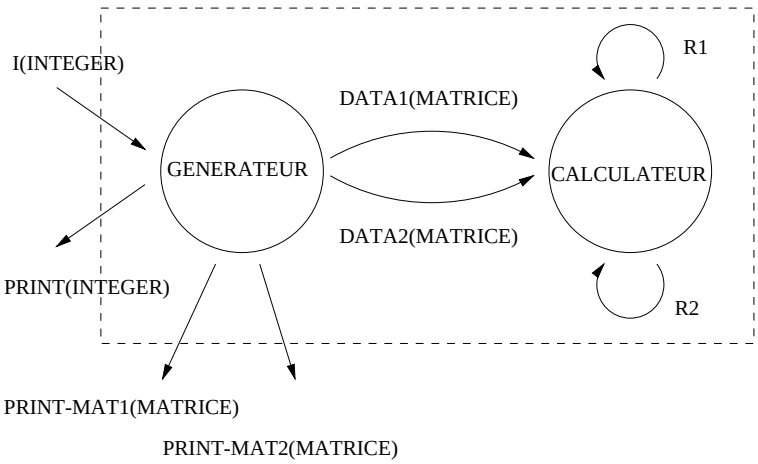

Figure 2.17: Exemple de l'additionneur de matrices

2.2.4 L'exemple de l'additionneur de matrices

Nous décrivons ici notre deuxième exemple d'application Esterel. Cet exemple nous permet d'illustrer le fonctionnement des tâches sous Esterel. Ce système est constitué de deux modules qui sont "générateur" et "calculateur". Le premier des deux tire aléatoirement deux matrices d'entiers qu'il affiche grâce aux deux signaux PRINT-MAT1 et PRINT-MAT2, le second fait la somme de celles-ci, puis affiche le résultat. On voit ici comment on peut écrire en Esterel une application qui manipule un type construit par l'utilisateur sans se préoccuper, dans un premier temps, de l'implantation du type et des opérations que l'on pourra effectuer sur celui-ci. Dans cet exemple, c'est le cas du type MATRICE où dans le source Esterel on ne spécifie pas le contenu des tâches startComputeMatrice et startAffichMatrice.

* **Le module générateur**

Dans ce module, on utilise les mêmes constructions que l'on avait utilisées dans l'exemple de l'émetteur/récepteur. On introduit ici deux notions supplémentaires importantes en Esterel : l'expression de la concurrence et l'utilisation de procédures et fonctions :

- L'instruction || permet d'exprimer le parallélisme entre des blocs d'instructions séquentielles délimités par les caractères [et]. C'est un parallélisme d'expression. Les différents blocs sont sérialisés à la compilation. L'exécution d'un automate Esterel est séquentielle,

- Esterel permet de définir des procédures et des fonctions. Les prototypes de ces procédures et de ces fonctions sont déclarés dans l'interface du module Esterel et c'est au programmeur de réaliser l'implantation dans le langage hôte.

* **Le module calculateur**

Ce dernier module s'occupe de l'addition des deux matrices. Celui-ci attend l'arrivée des signaux DATA1 et DATA2 (qu'il reçoit en parallèle), puis démarre une tâche startComputeMatrice qui additionne le contenu de DATA1 et DATA2 pour mettre le résultat dans "result". Ce résultat est alors affiché par une autre tâche qui est startAffichMatrice. Dans ce module, on a déclaré deux signaux internes un peu particuliers : ces signaux sont déclarés par l'instruction "*return*". Ils doivent être envoyés par la tâche pour signaler à l'automate que celle-ci vient de se terminer. Bien que la déclaration de ces signaux soit différente, ceux-ci sont gérés de la même manière que des

```
module calculateur:

type MATRICE;

return R1;
return R2;

task startComputeMatrice (MATRICE) (MATRICE,MATRICE);
task startAffichMatrice () (MATRICE);

input DATA2(MATRICE);
input DATA1(MATRICE);

loop
      var result : MATRICE in
            [
            await DATA1;
            ||
            await DATA2;
            ]
            exec startComputeMatrice (result) (?DATA1,?DATA2) return R1;
            exec startAffichMatrice () (result) return R2;
      end var
end loop
end module
```

Figure 2.18: Source Esterel de l'additionneur de matrices

signaux normaux, et en particulier, la technique décrite dans le paragraphe "Exécution de programme Esterel" est aussi celle utilisée pour les émettre depuis le langage hôte.

```
module generateur:

type MATRICE;

input I(integer);
output DATA1 (MATRICE);
output DATA2 (MATRICE);
output PRINT (integer);
output PRINT-MAT1(MATRICE);
output PRINT-MAT2(MATRICE);

function matriceAleatoire() : MATRICE;

loop
      await I;
      emit PRINT(?I);
end loop
||
every I do
      [
      var matrice : MATRICE in
            matrice:=matriceAleatoire();
            emit DATA1(matrice);
            emit PRINT-MAT1(matrice);
      end var
      ]
      ||
      [
      var matrice : MATRICE in
            matrice:=matriceAleatoire();
            emit DATA2(matrice);
            emit PRINT-MAT2(matrice);
      end var
      ]
end every
end module
```

Figure 2.19: Source Esterel du générateur de matrices

2.3 Validation de programmes synchrones

Bien que la validation de programmes Esterel ne fasse pas partie de cette étude, nous présenterons succinctement dans ce paragraphe la manière dont les preuves de programmes Esterel peuvent être conduites. Nous donnerons deux exemples d'outils utilisant chacun un fichier différent obtenu après compilation d'un programme Esterel : les fichiers OC[7] et les fichiers BLIF[8].

Un fichier OC contient les états d'un automate d'états finis[67, 47] correspondant à un programme Esterel. Lorsqu'un utilisateur souhaite vérifier un certain nombre de propriétés, il doit d'abord utiliser le compilateur Esterel pour obtenir le fichier OC de son application, puis faire appel à un outil dit "model-checker". TempEst[43] est un "model-checker" qui utilise un fichier OC en entrée. Il est basé sur une logique temporelle linéaire[13]. L'utilisateur spécifie en logique temporelle les propriétés de sécurité[49] qu'il souhaite vérifier (TempEst ne gère que les propriétés de sécurité et non celles de vivacité. Une propriété de vivacité ou "liveness" exprime intuitivement *"qu'une bonne chose arrivera éventuellement"*. Les propriétés de sécurité ou "safety" expriment pour leur part *"que de mauvaises choses ne seront jamais vérifiées"*, et ce quelles que soient les exécutions du programme). Cette spécification est traduite automatiquement par TempEst en un programme Esterel qui émet un signal quand une des propriétés de la spécification est violée. Le programme de l'utilisateur et celui généré par TempEst sont ensuite mis en parallèle. TempEst applique ensuite sur ce nouveau programme une technique de "model-checking" qui consiste à parcourir tous les états de l'automate afin de déterminer si l'un d'eux conduit à l'émission d'un signal détectant la violation d'une propriété. Cette technique de "model checking" est souvent appelée technique de "l'observateur".

Un fichier BLIF décrit l'automate d'une manière implicite (c'est à dire sans énumérer explicitement les états) à l'aide d'un système d'équations booléennes. Ce format est utilisé par l'outil CHECKBLIF[10]. Le fonctionnement de cet outil est proche de celui de TempEst. L'utilisateur écrit un programme observateur en Esterel qui émettra un signal lorsque la propriété sera violée. Il compile son programme Esterel avec l'observateur afin d'obtenir le fichier BLIF. Le fichier BLIF ainsi que le nom du signal détectant la violation de la propriété sont donnés en entrée de CHECKBLIF. CHECKBLIF recherche alors si le signal est émis dans l'espace des états de l'automate et, si c'est le cas, donne la suite des transitions pour arriver sur l'état fautif. Ce

[7] OC Pour **O**bject **C**ode.

[8] BLIF pour **B**erkeley **L**ogical **I**nterchange **F**ormat.

chemin peut alors être visionné par un outil comme XES[41]. Sur de grands programmes Esterel, le nombre d'états est très important. L'utilisation des fichiers OC est alors impossible. Le format BLIF à l'inverse, autorise les traitements sur de gros programmes. En effet, le format BLIF permet l'utilisation d'une représentation de l'espace des états qui économise de la place mémoire : les BDD[9][12]). Les BDD décrivent implicitement les états plutôt que de les énumérer. D'autres outils de preuves utilisent les fichiers BLIF et les BDD. Une partie du projet MEIJE[30] de l'INRIA[10] a pour objectif l'étude des environnements de vérification et d'analyse de systèmes communicants. Une équipe de ce projet travaille sur un outil de preuves nommé FCTOOLS[19] qui utilisent les BDD et les fichiers BLIF.

2.4 Conclusion

Esterel n'est pas seulement un langage de programmation. C'est un langage permettant d'effectuer des spécifications, de les prouver, et d'obtenir un prototype permettant de faire une première évaluation de l'application finale. Ce langage synchrone peut être associé à des méthodes de conception s'appuyant sur des représentations sous formes d'automates d'états finis. Nous avons vu que plusieurs outils de preuves existaient pour Esterel. Un certain nombre de ces outils ont été récemment associés pour construire un environnement complet de développement d'applications synchrones : la plate-forme SPORTS[11][41]. SPORTS rassemble, en plus des outils de preuves, des éditeurs graphiques ou textuels, des générateurs de code, et des outils de mise au point et de simulation. Néanmoins, dans le cadre d'applications embarquées, l'utilisation directe du code généré par le compilateur Esterel reste aujourd'hui encore difficile. Pour des programmes complexes où le nombre d'états des automates est important, la taille et les performances du code généré ne sont pas compatibles avec les contraintes des logiciels embarqués. Cependant, les travaux en cours sur l'optimisation d'Esterel permettent d'espérer que ces difficultés seront bientôt levées.

Dans ce chapitre, nous avons présenté le modèle synchrone fort, puis nous avons décrit le langage Esterel. Le modèle synchrone fort a été conçu pour un environnement centralisé, dans le chapitre suivant, nous allons exposer le modèle Saturne qui permet une exécution répartie de noyaux synchrones.

[9] BDD pour **B**inary **D**ecision **D**iagram.
[10] INRIA pour **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique.
[11] SPORTS pour **S**ynchronous **P**rogramming **O**f **R**eal **T**imes **S**ystems.

Chapitre 3

Le modèle Saturne

3.1 Le modèle synchrone faible

Dans le chapitre deux, nous avons décrit le modèle synchrone fort qui permet de spécifier des applications temps réel en bénéficiant à la fois d'un parallélisme d'expression, d'un comportement déterministe durant l'exécution et de possibilité de preuve logique et temporelle. Toutefois, ce modèle ne prend pas en compte les temps de communication. En effet, il a été conçu pour des systèmes centralisés, et non pour des systèmes distribués. Dans un environnement distribué, les temps de communication ne sont pas négligeables et ne permettent plus de supposer que les temps d'exécution et de communication d'un noyau soient nuls. Un autre modèle doit donc être utilisé. Il doit permettre de prendre en compte la répartition d'une application tout en conservant les propriétés prouvées dans le modèle synchrone fort. Il doit fournir une réactivité suffisante pour les applications temps réel, et rester aisément adaptable à une évolution de l'environnement d'exécution. Le modèle qui peut être utilisé ici est le modèle synchrone faible[1].

*** Définition du modèle synchrone faible:**

Dans ce modèle, il existe une horloge commune à tous les noyaux. Les noyaux ne réagissent plus par réflexe, mais périodiquement. L'horloge est une variable globale du système et les noyaux sont tous activés en même temps sur les tops de cette horloge globale. La notion d'instant d'activation s'apparente alors à cette activation périodique et non plus au moment où les signaux sont présents à l'entrée du noyau comme c'était le cas dans le modèle synchrone fort. Cet instant logique permet de limiter les temps de communication des signaux. **En d'autre termes, le réseau de communication doit être capable de garantir qu'une communication soit d'une du-**

rée bornée. Cette borne fait alors partie de l'intervalle entre deux tops de l'horloge globale : un noyau synchrone prend un intervalle de temps pour ses calculs et ses communications. On échange la notion de temps de calcul/temps de communication nul qui était spécifié dans le modèle synchrone fort par un temps de calcul/communication égal à **un instant logique de durée constante.** Cet intervalle de temps définit la période de l'horloge globale du système.

3.2 Le modèle Saturne

3.2.1 L'architecture de Saturne

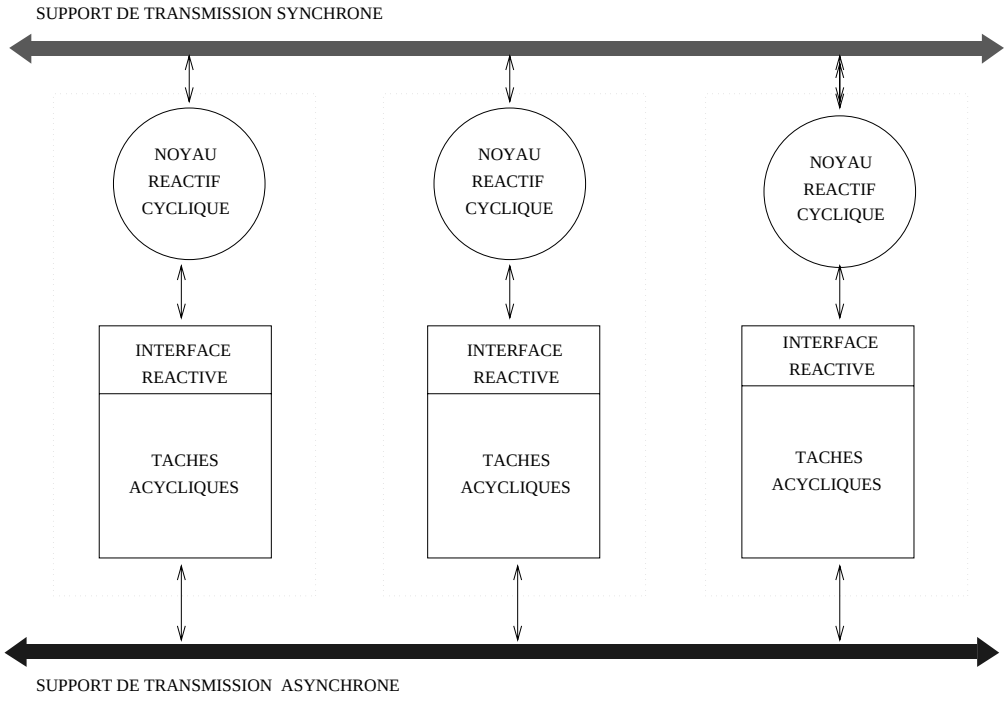

Figure 3.1: L'architecture Saturne

L'architecture du modèle Saturne[9, 8] correspond à un couplage entre modèle synchrone faible et modèle synchrone fort. Ce couplage permet de bénéficier de tous les avantages des deux modèles. Il confère à chaque noyau réactif un comportement externe périodique, et un temps de réaction de durée constante (donc compatible avec le modèle synchrone faible), tandis que son

comportement interne est du type réflexe instantané (donc synchrone fort). Saturne est constitué de grappes connectées entre elles par deux réseaux comme le montre la figure 3.1. Chaque grappe est constituée par :

- Un noyau synchrone : c'est la partie réactive de l'application. Elle est constituée d'un automate Esterel. Le noyau synchrone suit le modèle synchrone fort et effectue les changements de mode de fonctionnement de l'application,

- Une partie constituée de l'interface réactive et des tâches transformationnelles. Les tâches transformationnelles sont pilotées par le noyau synchrone. Toutefois, les noyaux synchrones ne commandent pas directement celles-ci, c'est l'interface réactive qui exécute les commandes du noyau synchrone. Les tâches transformationnelle constituent la partie calcul du système.

*** Notion de tâche interruptible**

Le modèle Saturne distingue deux types de tâches transformationnelles : des tâches conventionnelles et celles qui sont dites "interruptibles". Les tâches conventionnelles reçoivent des données en entrée et ne peuvent fournir un résultat **qu'à la fin de leur exécution.** Par notion de tâches interruptibles, on entend une tâche qui a la possibilité de fournir un résultat d'une qualité donnée à n'importe quel moment de son exécution. Ces tâches sont aussi appelée "*anytime*"[52]. En général, elles sont constituées de traitements itératifs qui fournissent un résultat de qualité croissante au fur et à mesure de leur exécution. C'est le cas des algorithmes d'optimisation que l'on peut trouver en recherche opérationnelle ou en intelligence artificielle. Une des difficultés de ce concept est de spécifier la qualité du résultat[1]. Par contre, l'avantage de ce type de tâche dans les applications temps réel est qu'elles fournissent une solution permettant de respecter les échéances de celles-ci. On peut ainsi concevoir l'utilisation dans des systèmes temps réel d'algorithmes dont la terminaison n'est pas bornée. **Toutefois, il faut noter que dans les applications d'avionique, l'utilisation des tâches transformationnelles est peu fréquente et que les tâches conventionnelles prédominent.**

*** Les primitives de Saturne**

Les tâches transformationnelles étant contrôlées par le noyau synchrone

[1]Quelle qualité de résultat doit on absolument obtenir une fois la tâche interrompue? Cette qualité sera t'elle suffisante?

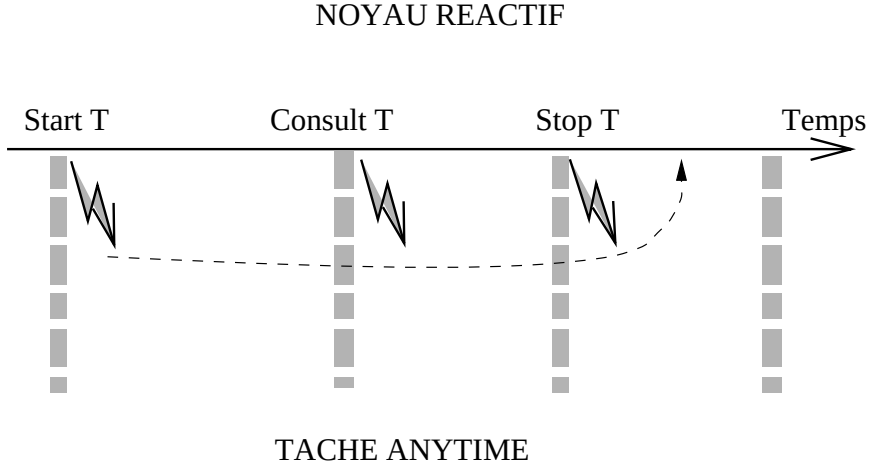

Figure 3.2: Utilisation des primitives de Saturne

du modèle Saturne, les commandes Saturne sont donc une extension du langage Esterel. En effet, les noyaux synchrones sont écrits en Esterel pour respecter le modèle synchrone fort. Les primitives Saturne sont au nombre de six :

- START_Ti [j] (paramètres en entrée) : activation de la tâche Ti avec les paramètres d'entrée. L'indice j est un numéro d'exemplaire : en effet, on peut imaginer que le noyau réactif ait besoin de démarrer plusieurs fois une même tâche avec des données différentes,
- KILL_Ti [j] : arrêt de l'exemplaire j de la tâche Ti sans récupération des résultats,
- CONSULT_Ti [j] : consultation des résultats courants de l'exemplaire j de la tâche Ti,
- STOP_Ti [j] : consultation des résultats courants de l'exemplaire j de la tâche Ti, puis arrêt de celle-ci,
- SUSPEND_Ti [j] : suspension de l'exemplaire j de la tâche Ti,
- RESUME_Ti [j] : réactivation de l'exemplaire j de la tâche Ti.

Sur la figure 3.2, on peut voir un exemple d'utilisation des primitives Saturne. Cette figure nous donne un chronogramme de l'exécution d'une tâche. Les tops d'horloge sont représentés par les barres verticales et l'envoi des commandes du noyau synchrone vers la tâche par les éclairs. La courbe

horizontale en pointillé montre l'exécution de la tâche. Celle-ci est lancée par une commande START au premier top puis, à chaque activation du noyau synchrone, le noyau consulte le résultat intermédiaire produit par la tâche transformationnelle. Au troisième top, le noyau décide d'arrêter la tâche et de récupérer le dernier résultat calculé. Cette action est réalisée par une commande STOP. L'arrêt de la tâche intervient avant le quatrième top.

* **Les réseaux de communication**

Le modèle Saturne génère deux modes de communication différents représentés sur la figure 3.1 par les deux flèches horizontales :

- Un mode synchrone pour les échanges entre noyaux réactifs. Dans ce mode, la taille des signaux est faible mais leur délai d'acheminement est très contraint car tous les signaux émis dans l'instant t doivent être reçus à l'instant $t + 1$,
- Un mode asynchrone pour les échanges entre tâches transformationnelles. Dans ce mode les messages échangés peuvent être de grande taille, mais les temps de communication sont moins critiques.

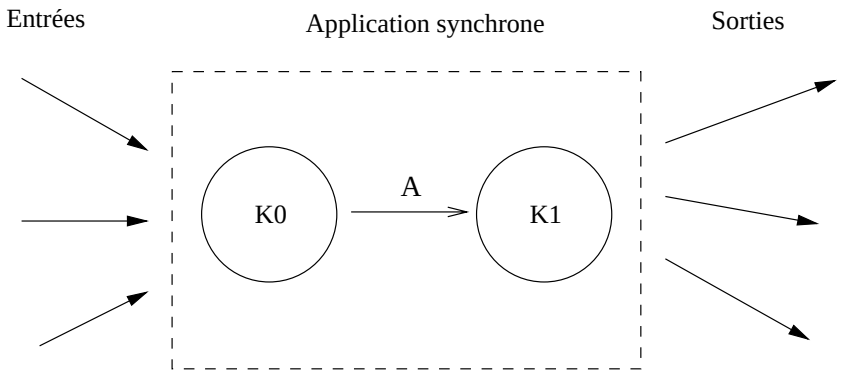

Figure 3.3: Le problème "50Hz-80Hz"

3.3 Extensions du modèle Saturne

Le modèle Saturne décrit ci-dessus a été conçu par l'ONERA-CERT entre 1992 et 1994. Depuis, plusieurs études ont été réalisées sur ce sujet. En collaboration avec Dassault Aviation, le CERT a fait évoluer le modèle Saturne pour l'adapter aux problèmes spécifiques des applications avioniques

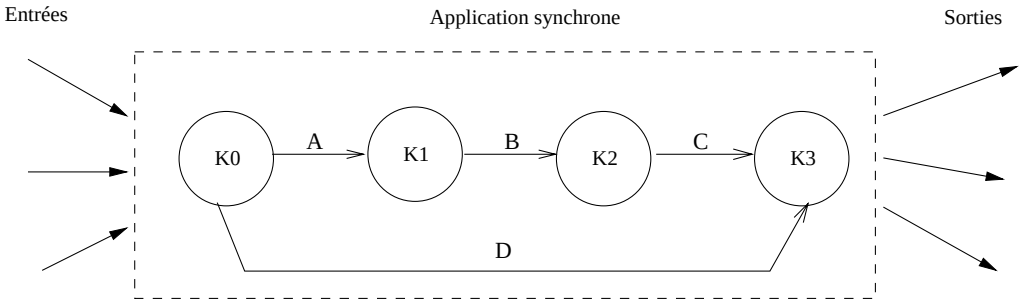

Figure 3.4: Le problème du "court-circuit"

embarquées. Nous présentons dans cette partie les problèmes posés par le modèle Saturne initial pour la modélisation d'applications embarquées, puis nous décrivons les extensions proposées par le CERT.

3.3.1 Comportements non modélisation avec Saturne

L'objectif initial de Saturne est la spécification d'applications temps réel embarquées. Or, un certain nombre de comportements courants dans ces systèmes ne sont pas directement modélisables avec la version initiale de Saturne. Voici deux exemples qui ont été détaillés dans un récent rapport du CERT[51] : le premier concerne la synchronisation de deux équipements qui sont activés à des fréquences différentes. Ce problème a été baptisé le problème "50Hz-80Hz". Dans les systèmes embarqués des avions, il existe de nombreux cas où deux équipements ont besoin de coopérer alors qu'ils ont des fréquences d'activations très différentes. La figure 3.3 nous montre deux noyaux synchrones. Le noyau K0 activé à 50Hz, envoie le signal A au noyau K1 qui est activé à 80Hz. Une telle application n'est pas modélisable en Saturne puisque Saturne active tous les noyaux en même temps grâce à son horloge globale. Le deuxième exemple est le problème du "court-circuit" signalé par E. Nassor et Y. Le Biannic, ingénieurs chez Dassault Aviation. La figure 3.4 nous montre quatre noyaux synchrones. Le noyau K0 émet deux signaux : A et D. Le noyau K3 reçoit les signaux C et D. La difficulté ici est de synchroniser l'arrivée des signaux C et D sur K3. Avec la première version de Saturne, on voit bien que si K0 émet A et D à l'instant t, D arrivera sur K3 en t+1 et C en t+3. Il faut donc pouvoir retarder suffisamment le signal D pour qu'il arrive sur K3 en même temps que C. Ce que le modèle Saturne initial n'est pas capable de faire.

3.3.2 Les extensions proposées : Saturne multi-synchrone

Pour résoudre les problèmes ci-dessus, le CERT a donc conçu un nouveau modèle : le modèle Saturne multi-synchrone[51] (nous parlerons de Saturne mono-synchrone pour le modèle Saturne sans les extensions décrites dans cette partie. **Il faut aussi préciser que les chapitres suivants de ce document concernent la version mono-synchrone de Saturne**).

Ce nouveau modèle reste assez proche de Saturne mono-synchrone. On y retrouve les grappes, les noyaux synchrones , les tâches transformationnelles, les interfaces réactives ainsi que les deux supports de communication. Les innovations concernent l'horloge globale de Saturne, les activations des noyaux ainsi que la communication des signaux Esterel. On y introduit de nouvelles abstractions :

- A chaque noyau synchrone on associe une horloge locale d'activation,
- On définit une horloge de référence. Cette horloge correspond à l'horloge globale de Saturne mono-synchrone. La fréquence de l'horloge globale est égale au plus petit commun multiple des fréquences de toutes les horloges locales des noyaux. Dans le modèle multi-synchrone, l'horloge de référence est toujours diffusée à tous les noyaux. Mais ceux-ci s'activent seulement quand le top de l'horloge de référence reçu correspond à un top de leur horloge locale. Cette correspondance peut se réaliser facilement par une opération de modulo entre l'horloge de référence et l'horloge locale. Ce nouveau mécanisme permet de modéliser des comportements de type "50Hz-80Hz",

$$\begin{cases} L_{recep,K1,A} = L_{recep,K2,B} = L_{recep,K3,C} = L_{recep,K3,D} = 0 \\ L_{emis,K0,A} = 0 \\ L_{emis,K0,D} = 2 \\ L_{emis,K1,B} = 0 \\ L_{emis,K2,C} = 0 \end{cases} \tag{3.1}$$

- On introduit enfin la notion de latence. Les temps de latence sont des retards qui sont appliqués à l'émission ou à la réception des signaux entre les noyaux. On définit donc une latence par signal pour chaque noyau synchrone. Nous noterons ici L_{recep} (respectivement L_{emis}) le temps de latence pour la réception (respectivement pour l'émission)

d'un signal. Ainsi, dans notre figure 3.4, le temps de latence en émission du signal A par le noyau K0 est noté $L_{emis,K0,A}$. Les temps de latence vont nous permettre de résoudre des problèmes de type "court-circuit". Le système d'équation 3.1 nous donne les valeurs des différentes latences pour que l'arrivée de C et D sur le noyau K3 soient synchronisées. Ici, on a choisi de positionner toutes les latences de réception à zéro. La synchronisation de C et D s'effectue sur les émissions : on positionne le temps d'émission de D à deux et tous les autres à zéro. En dehors du coupe-circuit, les temps de latence ont de nombreuses autres applications possibles, mais ils ont aussi quelques inconvénients. L'utilisation de plusieurs horloges et des latences complique singulièrement l'exécution d'une application Saturne. Il devient difficile d'en comprendre le fonctionnement. De plus, sur de très grosses applications, la détermination des latences devient vite compliquée. L'utilisation des latences nécessitera très certainement la réalisation d'outils permettant de positionner automatiquement des latences, ou au moins, de vérifier que leurs valeurs soient correctes par rapport à une spécification de la synchronisation voulue.

3.4 Le modèle Saturne Objet

Les systèmes embarqués sont compliqués et de taille importante. Il est donc nécessaire de pouvoir y associer des outils et des méthodologies. Le monde objet est riche dans ce domaine et un certain nombre de travaux ont tenté d'associer le monde synchrone au monde objet. F. Boulanger dans sa thèse[11] a intégré des noyaux synchrones dans des classes C++. Ainsi, il est possible d'appliquer des opérations issues du monde des objets sur des "objets synchrones", comme l'héritage, et surtout l'instanciation dynamique. Les travaux de Boulanger ont donnés lieu, entre autres, à une mise en oeuvre d'objets synchrones sur une plate-forme temps réel VxWorks. Les objets synchrones furent par la suite utilisés pour la définition d'un modèle Saturne objet par Faure[22], que nous allons maintenant rapidement décrire.

3.4.1 Description du modèle Saturne Objet

Le modèle Saturne Objet diffère du modèle Saturne initial principalement par deux aspects :

- La notion de grappe disparaît. Avec le modèle Saturne initial, une tâche était pilotée par un noyau et un seul. Toutes les tâches qui appartenaient à un noyau étaient regroupées ensemble. Ici, toutes les

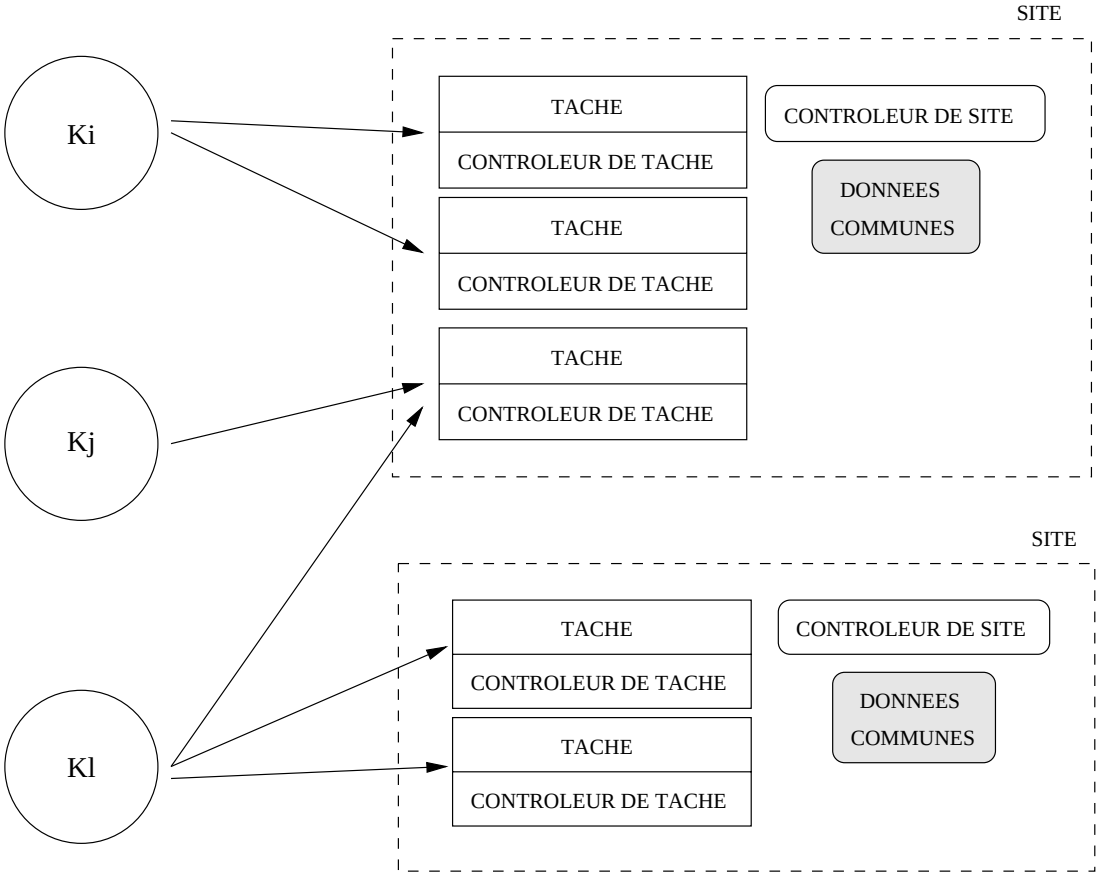

Figure 3.5: Le modèle Saturne Objet

tâches utilisant des données communes sont regroupées sur un même site. Les contrôleurs de site assurent la cohérence des données communes. Enfin, l'interface réactive est remplacée par un contrôleur de tâche. Chaque tâche est pilotée par un contrôleur de tâche (voir notre figure 3.5). Ce nouveau regroupement des tâches implique que les noyaux peuvent piloter des tâches de n'importe quel site. Toutefois, une tâche est toujours contrôlée par un noyau et un seul,

- Le système devient dynamique. Les composants de Saturne Objet sont pour certains, réalisés sous formes d'objets synchrones. Ils peuvent donc être instanciés durant l'exécution. C'est le cas des noyaux synchrones, des contrôleurs de site, des tâches et de leur contrôleur.

Il est difficile de dire si l'aspect dynamique de Saturne Objet est compatible avec des applications temps réel dur : ce modèle a été implanté dans le monde Unix, mais pas sur une plate-forme temps réel.

3.5 Contraintes apportées par le modèle Saturne sur l'environnement d'exécution

A partir des paragraphes précédents, on voit rapidement les contraintes qu'apporte le modèle Saturne mono-synchrone sur l'environnement d'exécution. Le système d'exploitation sur lequel Saturne peut fonctionner doit permettre :

- De supporter deux modes de communication par processeur **dont un qui soit synchrone**. Ceci est nécessaire pour respecter les contraintes du modèle synchrone faible. Bien que ceci ne soit pas standard dans CHORUS et que cette étude ne fasse pas partie de ce mémoire, certains travaux, comme ceux réalisés au CNET[2][62, 23], ont démontrés qu'il était possible de réaliser ce travail. Dans ce projet, une interface FDDI[3] cohabitait avec une interface Ethernet. Le coupleur FDDI supportait le trafic synchrone tandis que le coupleur Ethernet permettait le partage de fichiers,

- De garantir des temps d'exécution bornés et prédictibles. CHORUS étant un système d'exploitation temps réel, cette contrainte devrait être satisfaite,

- De résoudre les problèmes d'ordonnancement et de fournir des mécanismes de préemption afin d'assurer un niveau de réactivité suffisant.

- Enfin, bien que ça ne fasse pas partie de la spécification du modèle Saturne, il est impossible d'ignorer les contraintes de tolérance aux pannes : en effet, cette propriété est primordiale dans les applications temps réel qui seront développées sur Saturne. CHORUS permet la mise en place de ce type de mécanisme grâce, entre autre, à sa transparence à la localisation et à sa notion de groupe[4]. **La tolérance aux pannes est l'objet de ce document. Le chapitre suivant décrit les mécanismes de tolérance aux pannes habituellement**

[2]CNET pour **C**entre **N**ational d'**E**tudes en **T**élécommunications.

[3]FDDI pour **F**iber **D**ata **D**itributed **I**nterface.

[4]Bien que celle-ci soit relativement pauvre comparée à des outils comme Isis[15, 16] ou Horus[68, 54].

utilisés dans les systèmes distribués. Nous présentons ces mécanismes au travers de quelques exemples qui prennent en compte les contraintes de cette étude : l'utilisation d'un système à objets dans un environnement temps réel à contraintes strictes.

Chapitre 4

La tolérance aux pannes dans les systèmes distribués

Dans ce chapitre, nous effectuerons un tour d'horizon des techniques de tolérance aux pannes qui sont aujourd'hui utilisées dans les systèmes répartis. Ce vaste domaine a donné lieu à une quantité très importante de travaux. Nous nous contenterons, pour notre part, de citer quelques exemples qui devraient définir les principales techniques utilisées dans ce domaine, et qui sont :

- La redondance qui consiste à répliquer un composant sur plusieurs machines différentes,
- L'enregistrement sur un support stable de l'état d'un composant afin de le restaurer en cas de panne.

Notre étude se situant dans le cadre d'applications temps réel embarquées, nous avons utilisé, le plus souvent possible, des exemples issus de systèmes distribués temps réel. Puisqu'une des contraintes de notre étude est l'utilisation d'un système à objets pour le développement de nos prototypes, nous commencerons dans la première partie par une description de systèmes à objets qui offrent des mécanismes de tolérance aux pannes. La deuxième partie de ce chapitre sera dédiée aux systèmes utilisant des points de reprise. La troisième partie concernera les mécanismes de redondance. Enfin, nous finirons par la description d'un exemple de système temps réel utilisant à la fois la redondance et les points de reprise.

4.1 Exemples de systèmes à objets tolérants les pannes

Dans cette première partie, nous allons donner quelques exemples de systèmes à objets qui implantent des mécanismes de tolérance aux pannes. Nous présenterons d'abord les systèmes Isis et Horus. Bien qu'ils ne soient pas des systèmes à objets, ils sont utilisés par Garf et Electra. Ils permettent de maintenir une synchronisation entre des processus distribués. Nous décrirons ensuite Electra, puis, nous terminerons sur l'environnement Garf. Electra et Garf utilisent des mécanismes de redondance.

4.1.1 L'environnement Isis et le système Horus

*** Notion de vues de groupe**

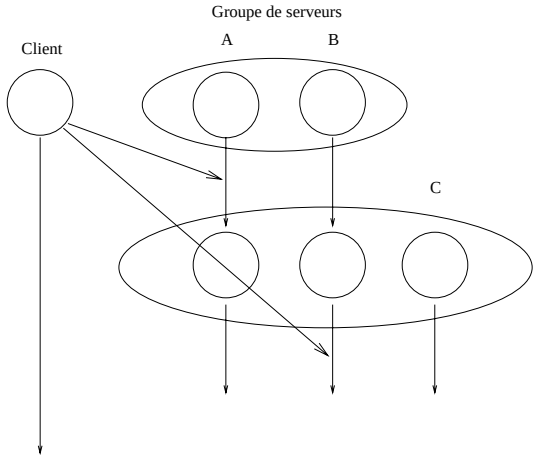

Figure 4.1: Le problème des vues de groupe

Nous avons évoqué dans l'introduction de ce chapitre qu'un des mécanismes de tolérance aux pannes les plus connu est la redondance. Lorsque l'on souhaite qu'un service soit maintenu lors de l'occurrence de pannes, on réplique le service sur d'autres machines. On définit ainsi la notion de groupe de serveurs. Un groupe est un ensemble de serveurs fournissant un même service. Il est alors intéressant pour le client de pouvoir invoquer d'une manière transparente tous les serveurs en même temps grâce à une diffusion sur groupe. Toutefois, si l'on se place dans le cas de groupes dynamiques, (un groupe est dynamique si l'on peut insérer et supprimer des serveurs dans

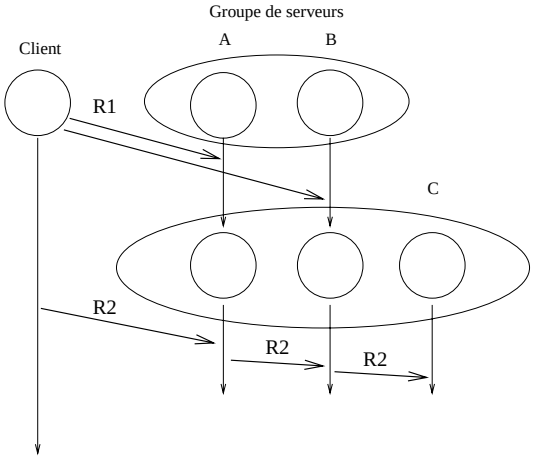

Figure 4.2: Le groupe est synchronisé virtuellement

le groupe pendant l'exécution de l'application), nous sommes confrontés à des problèmes de cohérence. En effet, dans les systèmes répartis, les temps de communication ne sont pas nuls (nous nous plaçons ici dans des systèmes où les communications sont asynchrones). Donc si aucun traitement n'est effectué, la connaissance qu'ont les clients de la constitution du groupe peut être erronée. On a besoin d'une vue cohérente du groupe.

Pour illustrer ce problème, regardons la figure 4.1. Dans cette figure, le client envoie un message vers un groupe de processus. Au moment où le client effectue sa diffusion, celui-ci pense que le groupe est constitué de A et de B. Pendant que A reçoit le message, C s'insère dans le groupe. C ne recevra pas de message du client alors que A et B le recevront. Le groupe est alors incohérent. Il faut donc maintenir chez **tous** les clients, une vue du groupe qui est cohérente avec la réelle composition de celui-ci. Si l'on prend l'exemple de la figure 4.2, on constate cette fois que l'évolution du groupe et des vues du groupe reste cohérente. La solution consiste à sérialiser les requêtes des clients et les demandes de modification du groupe. Ici, l'insertion de C est retardée jusqu'à ce que R1 soit terminé, puis R2 est retardé jusqu'à ce que C soit inséré : on dit qu'ils sont synchronisés virtuellement.

En dehors du problème de la cohérence des vues de groupe, se posent aussi des problèmes d'ordre sur les diffusions. Considérons un groupe de serveurs auquel sont envoyés des messages provenant de plusieurs clients différents. Pour que les états de tous les serveurs restent identiques, il faut que toutes ces diffusions arrivent à tous les serveurs, et dans un ordre identique.

Des protocoles sont donc nécessaires pour offrir des propriétés d'ordre et d'atomicité sur les diffusions dans les systèmes qui utilisent la redondance.

*** Une solution : Les environnements Isis et Horus**

Pour résoudre ces problèmes, on peut utiliser des outils comme Isis[15, 28, 54, 63, 16, 31] ou Horus[64] de l'université de Cornell (Mr Birman). Isis est une bibliothèque de protocoles et de services alors qu'Horus est une évolution d'Isis organisé comme un micro-noyau.

Ces outils permettent la mise en oeuvre d'algorithmes coopératifs et de mécanismes de tolérance aux pannes. Ils sont essentiellement basés sur quatre concepts :

- La notion de groupe de processus,
- Des protocoles de diffusion ordonnée.
- La synchronie virtuelle,
- L'atomicité des diffusions,

*** La synchronie virtuelle**

La notion de synchronie virtuelle a été introduite par Isis. On dit que les serveurs d'un même groupe sont synchronisés virtuellement si tous les processus membres voient arriver les événements au même moment sur les serveurs valides du groupe. Ceci doit être réalisé grâce à une diffusion atomique. Par serveur valide, on considère un serveur qui n'est pas en panne. Une défaillance temporaire du support de communication (une perte de message par exemple) n'est pas considérée comme une cause transformant un serveur valide en un serveur invalide.

La notion de diffusion atomique introduite ci-dessus correspond à la diffusion d'un message sur un groupe où l'on garantit que le message arrive à tous les serveurs. Si ce message ne peut arriver à destination d'un ou plusieurs serveurs, alors le protocole garantit qu'il n'arrive sur aucun serveur. Une diffusion atomique peut être réalisée par un protocole à deux phases.

La synchronie virtuelle permet de rendre les comportements des applications réparties prédictibles. Elle maintient la cohérence des traitements, simplifie le développement des applications réparties.

*** Protocoles de diffusion ordonnée**

Enfin, en plus de la synchronie virtuelle, Isis et Horus offrent plusieurs algorithmes permettant d'ordonner les diffusions :

- Le protocole FBCAST (FIFO[1] Broadcast), permet de respecter un ordre FIFO,
- Le protocole CBCAST (Causal Broadcast), garantit un ordre causal et atomique,
- Le protocole ABCAST (Atomic Broadcast). Celui-ci offre un ordre total et atomique,
- Le protocole GBCAST (Group Broadcast), offre un ordre total et sert de support pour la gestion des groupes.

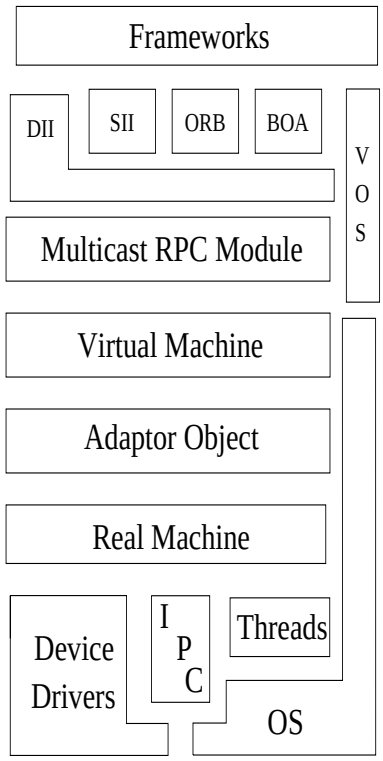

Figure 4.3: L'architecture d'Electra

[1]FIFO pour **F**irst **I**n **F**irst **O**ut.

4.1.2 Le système Electra

Electra[35, 61, 36, 37] est un bus à objets gratuit décrit dans la thèse de Silvano Maffeis[38]. Il implante une partie du standard CORBA[2], tout en offrant des fonctions supplémentaires tel que la communication entre groupes d'objets.

Electra est implanté au dessus d'Isis et Horus. Des versions sont prévues sur CHORUS[65] et Transis[40]. Electra a été spécialement conçu pour faciliter son adaptation sur d'autres systèmes. Dans cette partie, nous allons décrire l'architecture d'Electra, puis nous présenterons les services qu'il offre.

*** L'architecture d'Electra**

La figure 4.3 représente les différentes couches d'Electra. Electra doit pouvoir s'adapter à n'importe quel système d'exploitation. Toutefois, pour fonctionner, un certain nombre de services doivent être fournis par le système d'exploitation. Pour permettre cette portabilité, Electra introduit la notion de machine virtuelle. Cette machine virtuelle possède une interface comprenant tous les services nécessaires à Electra. C'est la couche d'adaptation qui permet de rapprocher la machine virtuelle de la machine réelle. Cette couche sera donc plus ou moins importante selon que le système d'exploitation fournisse ou non les services nécessaires. Ces services sont :

- Existence de processus légers ("*threads*"),
- Notion de groupe de processus, de vue de groupes et d'algorithmes de diffusions sur ces groupes,
- Outils de synchronisation.

Au dessus de cette machine virtuelle, Electra implante les outils qui vont être chargés de la communication entre les objets : c'est une couche d'appel de procédures à distance. Cette couche est capable de fournir des communications point à point mais aussi des communications de type "un vers n". La couche suivante fournit les services que l'on trouve habituellement dans CORBA. Enfin, la dernière couche d'Electra est constituée d'un ensemble de "*frameworks*" que nous étudierons rapidement à la fin de cette partie.

*** Les services offerts par Electra**

[2]CORBA pour **C**ommon **O**bject **R**equest **B**roker **A**rchitecture. Une description de CORBA peut être obtenue dans [21].

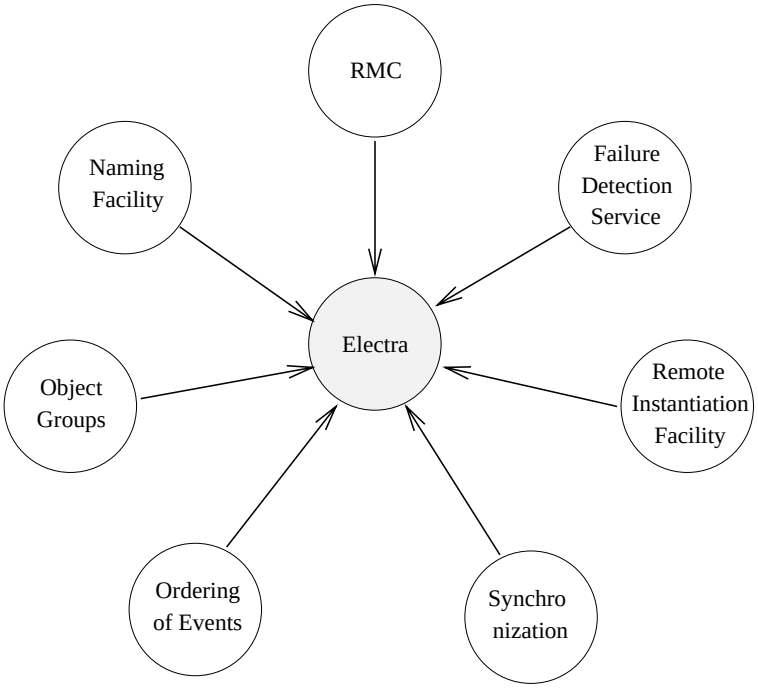

Figure 4.4: Les services d'Electra

Nous allons maintenant décrire les services qu'offre Electra aux développeurs d'applications. Ceux-ci sont tous résumés sur le graphe 4.4. Nous verrons successivement les mécanismes de communications utilisés, les algorithmes sur lesquels ils sont basés, la gestion des groupes, puis les services annexes.

* RMC (Remote Method Call)

Comme nous l'avons dit ci-dessus, l'appel de méthode à distance constitue le mécanisme de base de la communication sous Electra. Il est indépendant du système sous-jacent puisqu'il utilise les primitives de la machine virtuelle. Chaque serveur est multi-thread. Les invocations de clients vers les serveurs utilisent les trois modes standards d'invocations de CORBA :

- L'invocation synchrone. Le client est bloqué jusqu'à la fin de l'exécution de sa requête chez le serveur,
- L'invocation asynchrone. La requête est envoyée de manière asynchrone. Le client n'est pas bloqué durant le traitement du serveur,

- L'invocation synchrone différée. Le client fait son invocation, il est débloqué tout de suite mais il peut consulter la complétion de son invocation quand il le souhaite. Dès qu'il sait que sa requête est terminée, il peut récupérer ses résultats.

Tous ces modes d'invocations utilisent des algorithmes de diffusion ordonnée quand il s'agit de communications sur des groupes de serveurs.

*** Algorithmes de diffusion ordonnée**

Les protocoles de diffusion fournis par Electra sont basés sur les systèmes Isis et Horus qui constituent une partie des machines réelles. On retrouve donc les algorithmes suivants qui sont les mêmes que ceux d'Isis et que nous avons brièvement décris dans la partie précédente :

- Diffusion FIFO (FMCAST)
- Diffusion par ordre total (AMCAST)
- Diffusion par ordre causal (CMCAST)
- Diffusion par ordre causal atomique (ACMCAST)

*** Gestion des groupes**

La gestion des groupes est un aspect important d'Electra puisque c'est grâce à elle que les mécanismes de tolérance aux pannes sont possibles. Dans Electra, les groupes sont dynamiques. On peut donc insérer et supprimer des serveurs d'un groupe durant l'exécution d'une application. Tous ces serveurs doivent avoir une interface identique. La gestion des vues de groupe est une des difficultés de l'utilisation des groupes dynamiques. Electra utilise Isis pour résoudre ce problème. Trois cohérences sur les vues de groupe sont fournies par Electra :

- Une cohérence faible. Electra ne réalise aucun traitement pour la gestion de la cohérence,
- Une cohérence forte. Electra garantit que les modifications sur le groupe sont vues par tout le monde dans un ordre identique,
- Et enfin, une cohérence dite "hybride" qui offre un ordre causal sur la vue du groupe.

Le client a la possibilité de faire des invocations sur groupes de manière transparente ou non. Avec l'invocation transparente, le client n'a pas connaissance de la constitution du groupe (il ne connaît ni le nombre ni l'identité des serveurs). Il fait un appel de méthode et c'est Electra qui le diffuse sur tous les serveurs. Si le client opte pour une invocation non transparente, c'est à lui d'effectuer explicitement cette diffusion. Il est à noter que les modes d'invocations sur groupe sont les mêmes que dans les invocations point à point.

Une fois l'invocation réalisée, le client a la possibilité de récupérer les résultats ou les exceptions de plusieurs manières :

- N réponses exactement. L'utilisateur peut spécifier qu'il souhaite recevoir les 1,2 ou N premières réponses du groupe sur lequel il effectue l'invocation,
- Par comparaison. Cette technique est utilisée lorsque le programmeur désire récupérer la valeur la plus souvent retournée par les serveurs du groupe. Electra attend d'avoir reçu toutes les réponses des serveurs avant de retourner le résultat au client,
- Enfin par majorité. Dans cette option, Electra attend d'avoir reçu un nombre de réponses correspondant à la majorité des serveurs du groupe avant de renvoyer les résultats au client. Il est à préciser que si l'on spécifie cette option, la notion de majorité s'adapte à l'évolution du nombre de serveurs dans le groupe, et ce, grâce à la gestion des vues de groupe d'Electra.

*** Autres services offerts par Electra**

En dehors des services que nous venons de décrire, Electra fournit aussi un certain nombre de services annexes tel que :

- Des outils qui permettent d'instancier des objets à distance,
- Des mécanismes de détection de panne : les pannes sont transmises de la même manière que les vues de groupe,
- Un service de nommage qui permet d'adresser un objet serveur grâce à une chaîne de caractères,
- Et enfin, des outils de synchronisation qui sont essentiellement des sémaphores. Ces outils sont basés sur le système sous-jacent.

*** Les Frameworks d'Electra**

Pour terminer cette description d'Electra, nous allons énumérer les "*frameworks*" qui sont livrés avec Electra. La notion de "*framework*" introduite par Electra est identique à celle de CORBA : ce sont des services que les utilisateurs peuvent modifier ou utiliser tel quel et dont l'objectif est de fournir une base pour des applications génériques. Electra propose trois "*frameworks*" qui exportent leurs services sous forme d'interfaces IDL :

- Un modèle Cohorte/Coordinateur. Ce modèle permet la mise en oeuvre de mécanismes de tolérance aux pannes. Ici, les processus sont rassemblés en un groupe. Tous les processus peuvent effectuer les mêmes tâches mais un seul d'entre eux est actif: il s'agit du coordinateur. Quand le coordinateur tombe en panne, un autre coordinateur est élu parmi la cohorte. Le coordinateur envoie régulièrement son état à la cohorte afin de permettre un redémarrage rapide. Il s'agit de redondance passive,

- Des outils d'administration d'objets voisins du protocole SNMP[3]. Cette interface IDL permet d'obtenir des informations sur des objets Electra, de déclencher des alarmes lors de l'occurrence d'événements, etc,

- Un système d'information de type USENET avec des propriétés de tolérance aux pannes.

4.1.3 L'environnement Garf

L'environnement Garf[4][26, 42, 24, 53, 25] est un système à objets associé à une méthode de développement d'applications réparties. Cet environnement est issu de l'Ecole Polytechnique Fédérale de Lausanne.

Son objectif principal est de simplifier le développement des applications réparties en générant automatiquement les comportements distribués et les propriétés de résistance aux pannes. La mise en oeuvre des services de tolérance aux pannes est basée sur des groupes d'objets. Pour atteindre cet objectif, Garf sépare une application en deux composantes :

- Les objets fonctionnels. Ils constituent les objets effectuant les traitements des utilisateurs. Ces objets ne tiennent pas compte des aspects de répartition, de concurrence ou de tolérance aux pannes,

[3]SNMP pour **S**imple **N**etwork **M**anagement **P**rotocol.
[4]Garf pour **G**énération **A**utomatique de programmes **R**ésistants aux **F**autes.

- Les objets comportementaux. Ces objets décrivent les comportements face à la répartition et à la tolérance aux pannes. Ils sont organisés en bibliothèques. Ils sont généralement prêts à l'emploi, sauf si l'utilisateur a besoin d'un comportement particulier. Il peut à partir des bibliothèques existantes et grâce à l'héritage, créer d'autres objets comportementaux.

* La méthode de développement sous Garf

Nous allons brièvement décrire ici la méthode de développement associé à Garf. Lorsqu'on souhaite écrire une application sous Garf, on doit suivre les étapes suivantes :

- On commence par écrire les objets utilisateurs (c'est à dire les objets fonctionnels). On effectue les tests et la mise au point sans tenir compte des propriétés comportementales. Cette phase de mise au point est facilitée par le fait qu'elle est réalisée dans un environnement centralisé,

- On choisit une politique de résistance aux pannes, de concurrence, de persistance. On détermine les nouvelles synchronisations qu'introduit la répartition et on choisit les outils de synchronisation nécessaires pour leur réalisation (sémaphores ou lecteurs/rédacteur). Cette étape consiste à trouver les objets comportementaux correspondant aux politiques sélectionnées,

- En fonction des choix précédents, on dérive les objets fonctionnels avec les objets comportementaux qui s'y adaptent.

* Les objets comportementaux

La bibliothèque des objets comportementaux est organisée sous forme d'un arbre. Les objets implantent de nombreux comportements (comme les redondances actives et passives). Ils sont essentiellement de deux types :

- Les "encapsuleurs" : ils sont développés indépendamment de l'objet utilisateur. Leur rôle est d'intercepter les invocations des objets utilisateurs,

- Les "messagers" : Il en existe un par invocation. Ils sont utilisés pour les diffusions de groupe. Ils réalisent la communication physique et sont localisés sur le site client. Le messager peut être assimilé à un objet "*proxy*" CORBA.

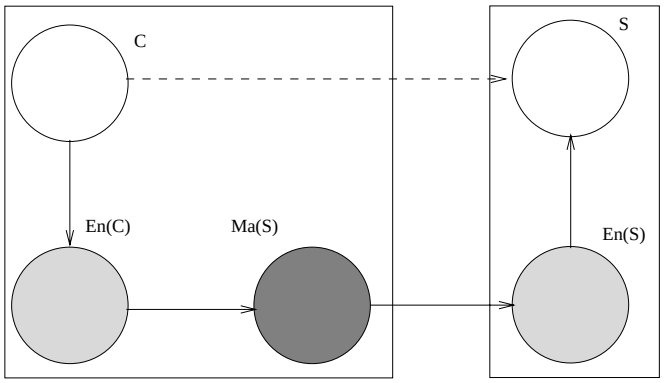

Figure 4.5: Mécanisme d'invocation sous Garf

La figure 4.5 représente une invocation de méthode sous Garf. L'objet client C souhaite appeler une méthode de l'objet S. L'appel de méthode est alors intercepté par Garf et c'est l'objet encapsuleur qui réalise l'invocation sur le messager. L'encapsuleur peut effectuer des traitements avant et/ou après l'invocation en fonction du comportement des objets client et serveur. L'invocation à distance est finalement faite par l'objet messager. Celui-ci est créé à chaque invocation. Du coté serveur, Il existe aussi un objet "*proxy*" qui effectue l'appel à l'objet serveur.

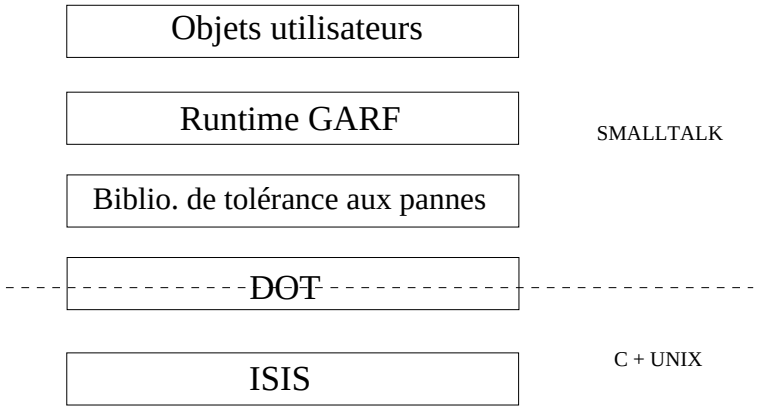

Figure 4.6: L'architecture de Garf

* **L'architecture de Garf**

Pour terminer cette introduction sur Garf, nous allons rapidement décrire son architecture. Celle-ci est représentée par la figure 4.6. Garf est construit au dessus d'Unix. Pour assurer les services de groupe, il s'appuie sur Isis. L'architecture de Garf peut être découpée en deux parties :

- La première partie est réalisée en C. Elle est constituée d'Isis et de la couche DOT[5]. Cette dernière a pour objectif de fournir aux couches supérieures les services d'Isis pour les objets. En effet les services d'Isis sont prévus pour des processus, or, chaque processus ne correspond pas à un objet,
- La deuxième partie constitue l'exécutable de Garf et les bibliothèques d'objets comportementaux. Cette partie est écrite en Smalltalk.

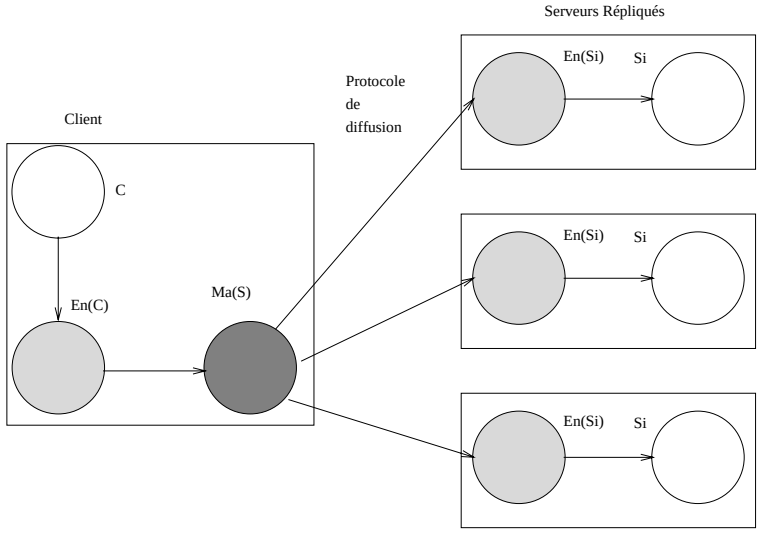

Figure 4.7: La diffusion sous Garf

* La diffusion sur groupe

Pour la réalisation des services de tolérance aux pannes, Garf se base sur des services de groupe offert par la couche Isis. Le protocole d'Isis utilisé par Garf est ABCAST. La diffusion sur groupe est transparente pour le client. Elle est essentiellement utilisée pour la mise en oeuvre des objets comportementaux implantant la redondance active. Un exemple d'invocation

[5]DOT pour **D**ependable **O**bject **T**oolkit.

est donné dans la figure 4.7. Comme dans une invocation point à point, l'invocation du client est interceptée par l'encapsuleur. Ici aussi c'est le messager qui s'occupe de la communication. C'est donc lui qui est en charge d'effectuer la diffusion sur le groupe.

* **Conclusion sur Garf**

Garf permet donc de développer des applications indépendamment de leur comportement face à la tolérance aux pannes. Cette séparation est réalisée grâce au langage de programmation objet, qui, par le biais de l'héritage, permet à un objet fils de bénéficier des propriétés comportementales de son objet père. D'autres systèmes utilisent aussi ce principe, on peut citer Avalon[20] ou Arjuna[60]. Certains systèmes vont plus loin dans cette approche. Ainsi [71] utilise un langage objet réflexif: Open C++. Dans ce langage, à chaque objet utilisateur est associé un objet appelé "méta-objet". Le programmeur peut influencer le comportement de l'objet utilisateur grâce au méta-objet. Il peut modifier la manière dont sont lus et écrits les attributs de l'objet utilisateur, modifier la manière dont sont invoqués les méthodes ou encore modifier le comportement lors de l'instanciation ou lors de la destruction d'un objet utilisateur. Comme pour l'héritage, les méta-objets sont utilisés pour implanter les mécanismes de tolérance aux pannes des objets utilisateurs.

4.2 Tolérance aux pannes par points de reprise

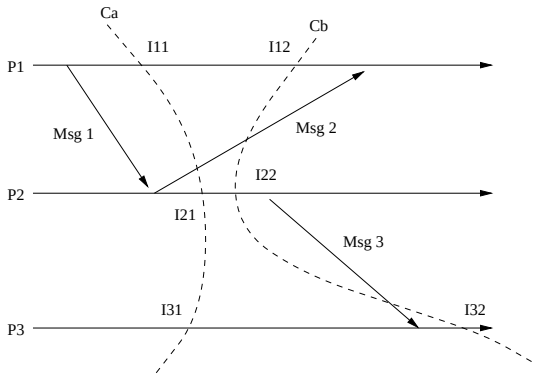

Figure 4.8: Exemple de coupe cohérente et de coupe non cohérente

Dans cette partie, nous allons décrire l'utilisation des points de reprise. Nous pouvons définir un point de reprise comme un ensemble de données permettant de reprendre le fonctionnement d'un processus qui s'exécutait sur une

machine tombée en panne. Un point de reprise représente l'état d'un processus. Les points de reprise des différents processus d'un système distribué doivent définir une coupe cohérente[56]. La figure 4.8 modélise un système distribué. Les droites P1, P2 et P3 représentent le fonctionnement de trois machines s'échangeant des messages. Nous considérons qu'un seul processus s'exécute par machine. Ces processus déclenchent des événements qui peuvent être des émissions ou des réceptions de messages. Une coupe est une partition en deux des événements du système : on sépare les événements futurs des événements passés. Une coupe cohérente est une coupe qui ne possède pas de réception de messages "*venant du futur*". La figure 4.8 nous donne deux coupes : *Ca* et *Cb*. Les intersections entre la courbe *Ca* et les droites *P1*, *P2* et *P3* constituent les dates des points de reprise. Ainsi les points I11, I21 et I31 sont les premiers points de reprise de P1,P2 et P3. La courbe *Cb* définit de la même manière trois autres points de reprise. La coupe définit par la courbe *Ca* est cohérente. La coupe définie par la courbe *Cb* ne l'est pas : en effet, le message *Msg 3* est un message du futur car son émission ne sera comptabilisée que dans le point de reprise I23 de P2 alors que sa réception est déjà comptabilisée dans le point de reprise I32 de P3. Si P2 tombe en panne et réexécute son programme à partir de I22, il émettra une deuxième fois le message *Msg 3*. Pour éviter ce type de problème, les points de reprise peuvent être réalisés par deux techniques :

- Tous les processus prennent en même temps les points de reprise. En cas de panne de l'un d'entre eux, tous les processus font marche arrière en même temps. La difficulté ici est de prendre des points de reprise qui correspondent à des états globaux cohérents. Des algorithmes tel que celui de K. Chandy et L. Lamport[32] permettent de définir ces états globaux cohérents en synchronisant les processus entre eux,

- Les processus prennent leurs points de reprise indépendamment les uns des autres. Si l'on n'enregistre que les points de reprise, il peut se produire le phénomène dit de "*domino*" . Lors d'une panne d'un processus, on relance le processus fautif sur une machine différente où il récupère son dernier point de reprise. Toutefois, le dernier point de reprise n'est pas toujours valide : en effet, il doit exister une coupe cohérente entre celui-ci et le dernier point de reprise de tous les autres processus. Si ce n'est pas le cas, alors il faut obliger les processus qui rendent non cohérente la coupe à effectuer une reprise sur un point antécédent, et ce, jusqu'à l'obtention d'une coupe cohérente : c'est l'effet *domino*. Pour régler ce problème, les processus enregistrent les messages sur disque et les rejouent lorsqu'ils sont relancés après une panne.

Les deux parties suivantes illustrent ces deux techniques : les objets de Clouds se synchronisent pour enregistrer leur états alors que les processus de GATOSTAR écrivent leurs points de reprise indépendamment les uns des autres.

4.2.1 Tolérance aux pannes dans Clouds

Le système Clouds[18] est un système distribué à objets. Il intègre des mécanismes de tolérance aux pannes. Il est constitué d'objets et d'activités. Les objets sont passifs et persistants, ils constituent chacun un espace d'adressage indépendant. Chaque objet est caractérisé par une capacité ne donnant pas sa localisation physique. Les objets peuvent d'ailleurs migrer pour des raisons de reconfiguration ou de répartition de charge. Les objets encapsulent les données et les programmes du système. L'accès à ces données s'effectue par l'appel de points d'entrée (c'est à dire par l'invocation de méthodes.). Le deuxième composant d'un système Clouds est l'activité (ou "*thread*"). Les activités "traversent" les objets par invocations successives. Plusieurs activités peuvent s'exécuter de manière concurrente dans un seul objet.

*** Mécanismes de tolérance aux pannes**

La tolérance aux pannes dans Clouds[2] repose sur des points de reprise sur les objets et les activités. La particularité de Clouds est la manière dont sont pris ces points de reprise. Clouds définit la notion de dépendance entre deux objets : un objet A est en dépendance avec un objet B si A invoque une méthode de l'objet B. Cette dépendance n'existe pas pour toutes les invocations : si la méthode invoquée est une méthode qui ne met pas à jour d'attributs sur l'objet serveur, alors la dépendance cesse d'exister lorsque l'invocation est terminée. Par contre, si l'invocation d'une méthode sur un objet modifie un attribut de celui-ci, alors la dépendance est maintenue après l'invocation. Les dépendances permettent de minimiser la taille des journaux sur disques. La prise d'un point de reprise est déclenchée par un objet initiateur. L'objet initiateur commence à enregistrer sur disque son état local qui contient entre autre les piles des activités qui exécutent ses méthodes, puis il consulte les dépendances qu'il a avec tous les objets du système et demande à chacun d'entre eux de faire de même. L'enregistrement des points de reprise est réalisé par un algorithme à deux phases. La première phase permet de savoir si tous les objets arrivent à enregistrer leur point de reprise, la deuxième phase est l'enregistrement à proprement dit. Ces mécanismes permettent d'obtenir des points de reprise cohérents pour tous les objets du système.

4.2.2 Le système GATOSTAR

L'objectif de GATOSTAR[55] est la répartition de charge et la tolérance aux pannes pour des applications parallèles distribuées à gros grains et à temps de calcul long. La plate-forme d'exécution cible de GATOSTAR est un réseau de station de travail sous Unix. Dans ce type de système, les pannes sont rares : des études ont estimé à 2,7 jours la fréquence moyenne de pannes sur un réseau local de dix machines. Toutefois, sur des calculs parallèles qui peuvent durer plusieurs jours, voir plusieurs semaines, cette probabilité n'est plus négligeable. Le système GATOSTAR est composé de deux parties : GATOS qui effectue la répartition de charge et STAR qui met en oeuvre les mécanismes de tolérance aux pannes. L'association de ces deux composantes se justifie par le fait que certains des services nécessaires à ces outils sont communs.

Les mécanismes de tolérance aux pannes de GATOSTAR s'appuient sur l'écriture de points de reprise et une journalisation des messages échangés entre les programmes parallèles. Dans GATOSTAR, les processus prennent leur point de reprise indépendamment les uns des autres. Les mécanismes de détection de pannes de GATOSTAR supposent que les processeurs soient "*fail-silent*". Les processeurs "*fail-silent*" sont des processeurs qui n'émettent aucun message lorsqu'ils tombent en panne et qui deviennent silencieux. La détection de panne peut être réalisée de deux façons :

- De manière passive : on attend qu'un processus qui a besoin de la machine en panne détecte la panne,
- De manière active : on scrute régulièrement toutes les machines pour connaître celles qui sont défaillantes.

GATOSTAR utilise ces deux techniques. La gestion des pannes dans GATOSTAR est réalisée de la façon suivante : les processus enregistrent régulièrement des points de reprise. Ils maintiennent aussi un journal contenant tous les messages reçus depuis le dernier point de reprise. Une fois la panne détectée par GATOSTAR grâce aux méthodes que nous avons citées ci-dessus, le processus est relancé sur une autre machine en fonction de la charge et des pannes. La reprise consiste à charger le dernier point de reprise et relancer l'exécution du processus. Les messages sont lus à partir du journal. Si le processus émet des messages, ceux-ci sont éliminés grâce à l'utilisation d'estampilles, puis, lorsque le processus arrive au niveau où la panne s'est produite, les messages qu'il émet ne sont plus écartés et son exécution continue normalement. La panne est alors recouverte.

Pour conclure, nous pouvons dire que l'inconvénient essentiel de GATOSTAR est son besoin important de ressources disque. En effet, la quantité de messages sauvegardés est très importante et la duplication de ces journaux augmente encore leur volume total. A l'heure actuelle, GATOSTAR fonctionne sur des machines Sun. Un portage est en cours sur Linux mais l'objectif de l'équipe est de pouvoir étendre GATOSTAR à la gestion de réseaux hétérogènes où cohabiteraient des stations de travail avec des machines multiprocesseurs.

4.3 Tolérance aux pannes par redondance

Nous commencerons cette partie par la description de Delta 4. Ce système est certainement le plus populaire des systèmes effectuant de la redondance. Il nous permettra d'introduire quelques définitions couramment utilisées dans la tolérance aux pannes. Ensuite, nous étudierons un deuxième système très connu : MARS. Enfin, nous finirons par HARTS.

4.3.1 L'architecture Delta 4

L'objectif de Delta 4 est de fournir un système qui facilite la conception d'applications distribuées tolérant les pannes. Un système Delta 4 est un ensemble de machines connectées par un ou plusieurs réseaux locaux. Deux architectures Delta 4 ont été définies :

- Delta 4 OSA[6][69, 70, 58] qui supporte des équipements hétérogènes sans contrainte de temps. C'est un système ouvert,
- Delta 4 XPA[7][59] qui possède des notions d'échéances et de priorités. XPA a moins de fonctionnalités qu'OSA mais est adapté aux systèmes temps réel.

Une application fonctionnant sur Delta 4 est un ensemble de composants indépendants s'exécutant sur un réseau de machines et communicant par messages. Ces composants sont répliqués sur plusieurs sites différents afin de faire face à des pannes de machines. Delta 4 offre trois types de redondances :

- La redondance active (voir figure 4.9). Dans ce mode, lorsqu'un message est transmis à un composant, toutes ses répliques le reçoivent,

[6]OSA pour **O**pen **S**ystems **A**rchitecture.
[7]XPA pour e**X**tra **P**erformance **A**rchitecture.

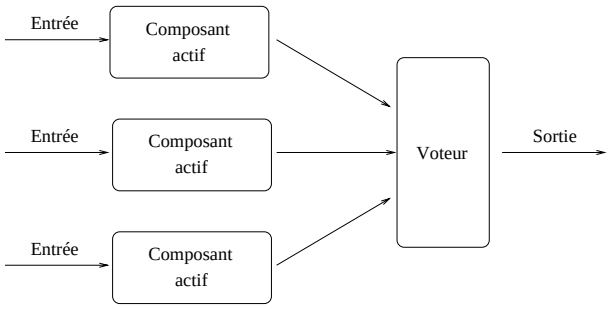

Figure 4.9: La redondance active

effectuent les calculs déclenchés par ce message, et émettent les résultats obtenus. Le résultat du composant est alors déterminé par un vote majoritaire entre les résultats de toutes les répliques. Le vote majoritaire consiste à considérer comme juste le résultat renvoyé par le plus grand nombre des répliques. Il faut noter que ce mode nécessite l'utilisation d'algorithme de diffusion atomique avec un ordre total afin de synchroniser les messages en entrée des différentes répliques. Ce mode est adapté aux systèmes temps réels. En effet, en cas de panne, le recouvrement est immédiat tant qu'il existe une ou des répliques valides. Nous verrons en page 59 que le nombre de répliques nécessaires dépend du comportement de la machine et du type de pannes considérés,

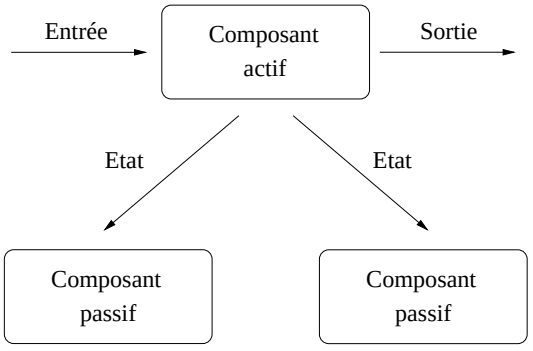

Figure 4.10: La redondance passive

- La redondance passive (voir figure 4.10). Ici, une seule réplique est active. Lorsqu'un message est transmis au composant, la réplique active le reçoit, réalise les calculs correspondant à ce message, puis émet ses résultats. Il n'y a donc pas de vote puisqu'un seul résultat est produit. La réplique active transmet régulièrement son état à toutes les répliques passives. En cas de panne de la réplique active, une réplique passive devient active,

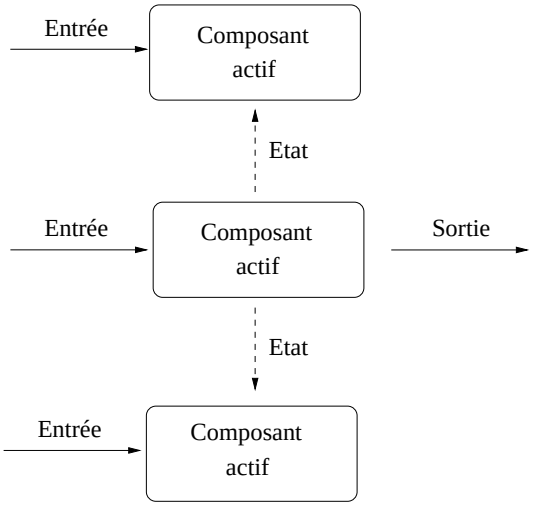

Figure 4.11: La redondance semi-active

- La redondance semi-active (ou modèle "*leader/followers*", voir figure 4.11). Ce dernier mode est un mélange des deux précédents. Toutes les répliques sont actives : elles reçoivent toutes les messages destinés au composant. Néanmoins, une seule des répliques retourne le résultat des calculs activés par le message d'entrée (le "*leader*"). Puisque toutes les répliques effectuent les calculs, elles ont toutes un état identique. Toutefois, un transfert de l'état du "*leader*" vers les autres répliques peut être déclenché quand un calcul est indéterministe.

Le choix entre ces trois techniques s'effectue essentiellement selon trois critères (voir tableau 4.1):

- Le comportement du composant. Si le composant est indéterministe, alors la redondance active ne peut pas être utilisée. En effet, pour un même message, si les répliques fournissent des résultats différents car

Redondances	Coût de traitement du mécanisme	Indéterminisme	Fail uncontrolled
active	la plus faible	interdit	autorisé
passive	la plus importante	autorisé	interdit
semi active	faible	autorisé	interdit

Tableau 4.1: La redondance sous Delta 4

leur algorithme est indéterministe, il est impossible d'appliquer un vote majoritaire. Par contre, l'indéterminisme du composant n'interdit pas l'utilisation de la redondance passive. Quant à la redondance semi-active, elle peut être utilisée si un transfert de l'état du "*leader*" vers toutes les autres répliques est effectué après l'exécution de la partie de l'algorithme qui est indéterministe,

- Le comportement des sites. On peut classer les sites en deux catégories selon leur comportement : les sites dit "*fail-silent*" et les autres dit "*fail-uncontrolled*". **Les sites "*fail-silent*" n'émettent aucun message lorsqu'il tombe en panne et deviennent silencieux. Les sites "*fail-uncontrolled*" quant à eux, peuvent émettre des messages corrompus lorsqu'une anomalie de fonctionnement intervient. Ils peuvent aussi envoyer des messages supplémentaires ou en perdre certains.** Dans le cas d'un système où les sites sont "*fail-uncontrolled*", l'utilisation de la redondance passive ou semi-active est interdite. Par contre, la redondance active peut être utilisée dans les deux cas : dans un système "*fail-uncontrolled*", le nombre de répliques actives nécessaires pour recouvrir t pannes de site est de $2t+1$. Le vote majoritaire permet alors de ne pas prendre en compte les messages considérés comme faux. Dans un système où les sites sont tous "*fail-silent*", $t+1$ répliques suffisent pour recouvrir t pannes,

- Le type de l'application. Si les composants constituent une application ayant des contraintes temporelles fortes, la redondance active est plus adaptée car elle nécessite très peu de temps pour recouvrir une panne.

4.3.2 Le système MARS

*** <u>Présentation et architecture</u>**

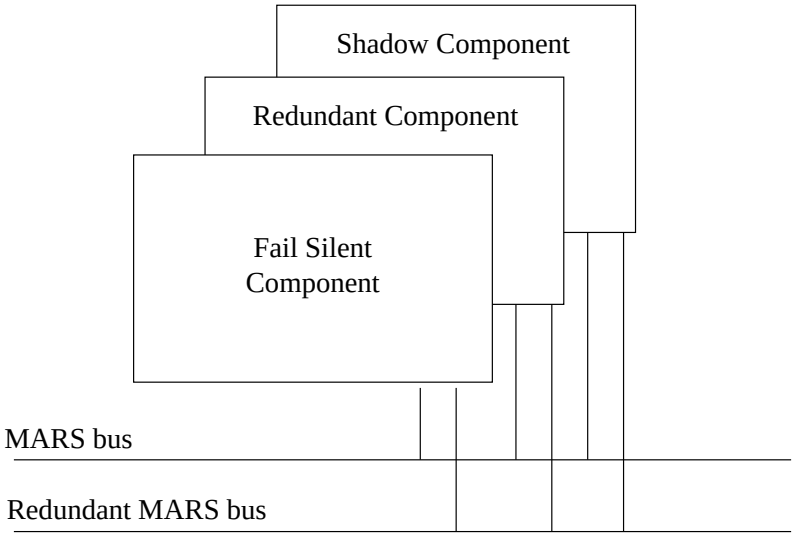

Figure 4.12: L'architecture de MARS

MARS[8] [57] est une architecture distribuée tolérante aux pannes pour les applications temps réel critiques. Le système est complètement déterministe. La figure 4.12 nous présente son architecture. MARS est constitué d'un ensemble de composants autonomes faiblement couplés par deux bus cycliques. Chaque composant est connecté aux deux bus. Les bus sont partagés entre les différents composants grâce à une politique de TDMA[9]. Chaque message émis par un composant est diffusé à tous les autres. Tous les composants possèdent une horloge locale synchronisée à l'aide d'un algorithme distribué de synchronisation d'horloge qui est tolérant aux pannes. L'horloge en question permet le partage des bus par TDMA. Les composants sont dupliqués par une politique de redondance active. Nous verrons plus loin qu'une troisième réplique existe (la réplique "ombre"), toutefois son fonctionnement est différent des deux autres.

* Mécanisme de tolérance aux pannes

MARS est capable de détecter et de recouvrir les pannes permanentes et les pannes transitoires de composants. Il s'attache principalement aux pannes matérielles. Les composants sont "*fail-silent*". Ils fournissent un résultat juste ou rien d'autre. Des batteries de tests leurs permettent de déterminer s'ils sont dans un état de bon fonctionnement ou non. Lorsqu'un

[8]MARS pour **MA**intainable **R**eal-time **S**ystem.
[9]TDMA pour **T**ime **D**ivision **M**ultiple **A**ccess.

composant détecte qu'il a un fonctionnement anormal, il s'arrête de lui-même. Ce principe permet de limiter les effets de dominos : un composant en erreur n'entraîne pas l'erreur d'autres composants.

La détection des pannes permanentes est réalisée par deux techniques :

- Par des tests effectués par le matériel,
- Par le système d'exploitation (grâce à des assertions, par le contrôle du temps d'exécution des tâches, etc.)

Ces méthodes ne sont toutefois pas applicables aux pannes transitoires. Une double exécution des tâches est utilisée pour les détecter. L'ordonnancement de MARS est statique, on connaît donc une borne maximale du temps d'exécution d'une tâche et en général, le temps prévu pour l'exécution de celle-ci permet dans 90 % des cas de la réexécuter une deuxième fois. Plutôt que de laisser le processeur oisif durant ce temps, on relance la tâche à la suite de sa première exécution. Si cette deuxième exécution se termine à temps, on compare leurs résultats et leurs temps d'exécution. En cas de différence, le composant se met à l'arrêt. Une différence sur les temps d'exécution des tâches signifie que la plus longue des deux a été victime de pannes transitoires.

Le recouvrement d'une panne est différent selon qu'il y a déjà eu une panne sur une des deux répliques actives ou non. Dans le cas où aucune panne n'est intervenue, les services fournis par le composant en panne sont toujours assurés grâce à la deuxième réplique active. Si une des répliques actives étaient déjà en panne, les services offerts par le composant ne peuvent plus être assurés. Le rôle de la troisième réplique, appelé réplique "ombre" est de parer à cette éventualité. En effet, quand l'une des deux répliques tombe en panne, la réplique "ombre" la remplace alors. En temps normal, le fonctionnement de la réplique "ombre" est identique aux deux autres en dehors du fait qu'elle n'émet aucun message sur le bus. Aucun slot TDMA ne lui est attribué pour qu'elle puisse faire ses émissions. L'état de la réplique "ombre" est pourtant identique aux deux autres car elle peut lire les messages provenant du bus. Elle effectue donc les mêmes calculs que les deux autres. C'est un mécanisme identique à la redondance semi-active de Delta 4.

Un service de MARS permet de notifier à toutes les répliques celles d'entres elles qui sont en pannes. Ce service donne à l'instant $t_{présent}$ toutes les répliques actives à l'instant $t_{passé}$. La différence entre $t_{passé}$ et $t_{présent}$ est de deux cycles TDMA maximum. Après une panne, les répliques fautives

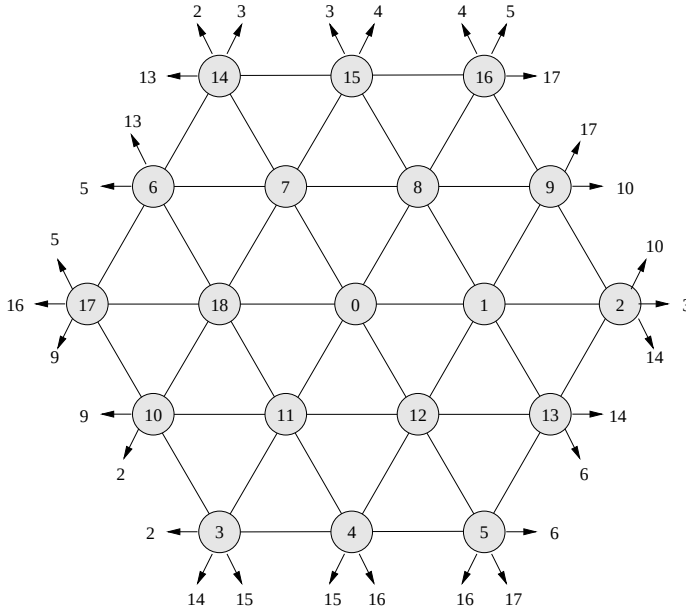

Figure 4.13: L'architecture de HARTS

tentent de se réinitialiser. Si leurs tests concluent qu'elles peuvent à nouveau être utilisées, elles rechargent leur programme et initialise leur état en copiant l'état courant d'une réplique qui fonctionne. Pour ce faire, il existe dans le système un composant dédié qui diffuse sur le bus les programmes de tous les composants. De plus, chaque réplique en fonctionnement diffuse régulièrement son état. Si deux répliques sont en cours de fonctionnement, la réplique réparée devient la réplique "ombre". S'il ne reste plus qu'une seule réplique en fonctionnement, la réplique réparée utilise le slot TDMA libre et devient une réplique de redondance active. Le même mécanisme est utilisé lorsqu'une réplique en panne est remplacée lors d'une phase de maintenance.

4.3.3 L'architecture HARTS

HARTS[10][29] est une architecture pour l'exécution d'applications temps réel distribuées. La figure 4.13 montre que HARTS est constitué d'un ensemble de noeuds connectés entre eux pour former un hexagone. Chaque noeud est une machine multiprocesseur qui possède six voisins. Les communications sont des communications point à point bidirectionnelles. Un noeud (voir figure 4.14) est constitué :

[10] HARTS pour **H**exagonal **A**rchitecture for **R**eal **T**ime **S**ystems.

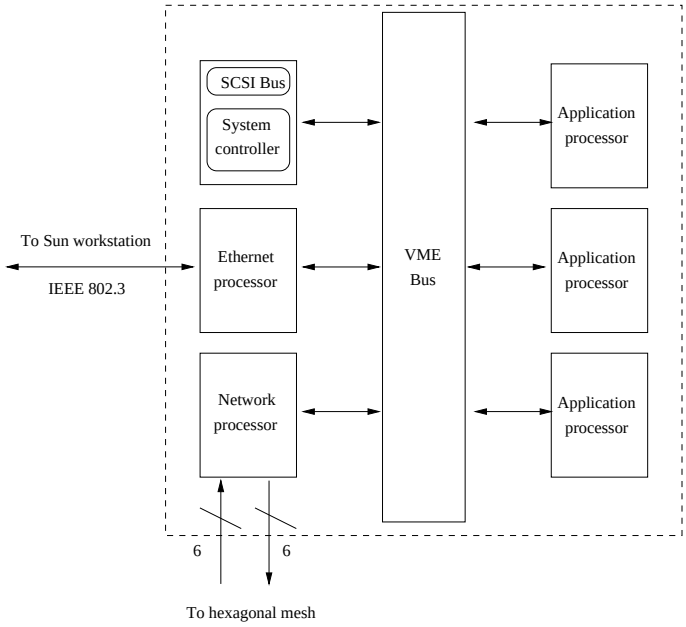

Figure 4.14: Un noeud HARTS

- De plusieurs processeurs où fonctionnent les applications,
- D'un processeur pour gérer la connexion avec le réseau hexagonal,
- D'un coupleur Ethernet pour pouvoir dialoguer avec une machine hôte,
- Et d'interfaces pour des disques de stockage. Le tout est connecté par un bus VME[11].

* Les mécanismes de tolérance aux pannes

Sur chaque processeur de HARTS fonctionne une version modifiée du système d'exploitation pSOS. Les services offerts par HARTS sont principalement :

- Un service de nommage distribué implanté dans chaque processeur réseau,
- Des outils de communication interprocesseurs,
- Une horloge globale, des canaux de communication fiable et temps réel avec garantie de délai de communication,

[11] VME pour VERSAmodule Eurocards.

- Des mécanismes de diffusion sur groupe et des mécanismes de tolérance aux pannes.

HARTS offre un certain nombre d'outils permettant de tester les applications : on peut charger une application sur HARTS puis simuler des fautes matérielles ou logicielles qui peuvent être transitoires, permanentes ou intermittentes. Le comportement de l'application peut alors être consulté grâce à des outils d'observation. Enfin, des outils de réexécution peuvent aussi être utilisés pour mettre au point l'application.

Les mécanismes de tolérance aux pannes introduits dans HARTS sont liés à son architecture d'interconnexion des noeuds. En effet, l'architecture hexagonale a été choisie pour permettre de faire face à des pannes de noeuds ou de liens tout en essayant de maintenir des connections point à point avec des bornes sur les temps de communications. Les techniques de communication utilisées dans le réseau hexagonal sont :

- Un mécanisme de diffusion. L'émetteur diffuse ses messages à ses voisins. Ceux-ci sont transmis par une vague inondante jusqu'au récepteur. Le récepteur peut donc obtenir plusieurs exemplaires d'un message sur lesquels il effectue un vote majoritaire. Le vote permet de supprimer les messages corrompus par des noeuds bizantins,

- Des canaux temps réel fiables et tolérants aux pannes. Une tâche, avant qu'elle souhaite communiquer avec une autre, s'adresse à un gestionnaire réseau pour allouer un canal entre lui et son destinataire. L'allocation des canaux est centralisée, toutefois, le gestionnaire de réseau est dupliqué (un état cohérent des répliques est maintenu par des diffusions ordonnées.)

4.4 Un exemple de système temps réel utilisant la redondance et les points de reprise

Nous avons présenté dans les parties précédentes des systèmes utilisant la redondance ou les points de reprise. Nous allons présenté ici le système COSMOS[12][3] qui est un système temps réel combinant l'utilisation de ces deux techniques. COMOS est développé conjointement par la NASA et le Centre de recherche de Langley. C'est un système temps réel distribué tolérant aux pannes. Les applications s'exécutent selon un modèle flot de données

[12]COSMOS pour **CO**mmon **S**paceborne **M**ulticomputer **O**perating **S**ystem.

décrit dans [21]. Ici, le flot de données est un flot de données à gros grains : chaque noeud est un programme séquentiel en C ou en ADA. COSMOS est conçu pour fonctionner sur une architecture distribuée où les machines sont connectées par deux bus :

- Un bus à diffusion fiable pour la synchronisation et les messages de contrôle,
- Un réseau utilisateur pour le transfert de données de calcul entre les différents processeurs.

Chaque processeur exécute un noyau VRTX : COSMOS est un logiciel de couche supérieure (un portage sur VxWorks est en cours).

*** Les mécanismes de tolérance aux pannes**

Les mécanismes de tolérance aux pannes introduits par COSMOS sont basés sur une redondance active des noeuds du graphe ainsi que par l'enregistrement de points de reprise. Une application COSMOS est constituée d'une partie statique qui n'évolue pas durant son exécution et d'une partie dynamique. La partie statique correspond à la structure du graphe : les noeuds et les arcs. La partie dynamique d'une application correspond aux "jetons" sur les arcs du graphe. Ces jetons sont des informations produites par des noeuds du graphe, c'est à dire par des programmes. Ils sont stockés sur les sites qui les ont calculés et transmis si nécessaire vers d'autres processeurs. Quand toutes les entrées sont présentes sur un noeud, une tâche est créée pour calculer les données en sortie. Plusieurs tâches calculant un même noeud peuvent s'exécuter en parallèle si les entrées nécessaires sont présentes. Lorsqu'un processus se termine, le site hébergeant la tâche diffuse sur le bus de contrôle les jetons qu'il a produit. Tous les processeurs reçoivent ce message et regarde quels sont les noeuds suivants qu'ils peuvent activer. Un algorithme d'élection choisit alors le site qui va exécuter les processus correspondant aux noeuds calculables. **Cette diffusion permet de maintenir une vue cohérente de l'état dynamique du graphe flot de données.** Toutefois, seule la présence des jetons et leur nombre par arcs sont connus par tous, les résultats des calculs restent sur les sites où ils ont eu lieu. Ils sont transférés par le réseau utilisateur en cas de nécessité. La cohérence offerte par le bus à diffusion permet la prise de points de reprise simultanée par tous les noeuds.

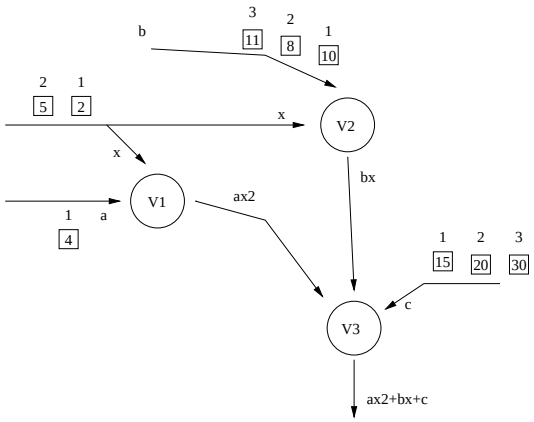

Figure 4.15: COSMOS : phase initiale

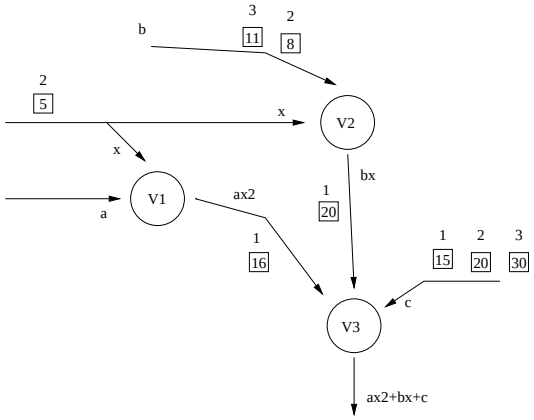

Figure 4.16: COSMOS : une bonne synchronisation

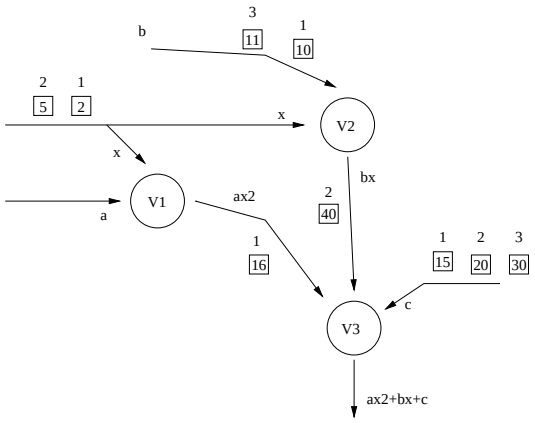

Figure 4.17: COSMOS : une mauvaise synchronisation

Les trois exemples 4.15, 4.16 et 4.17 nous montrent que les données sur les arcs sont numérotées. La numérotation des jetons est utilisée car :

- Elle permet de déterminer quels sont les processus qui peuvent être démarrés : si toutes les entrées d'un même numéro sont présentes à un noeud, un processus peut être instancié. La figure 4.15 correspond à la phase initiale de notre exemple. On voit que l'on peut instancier un processus sur V1 et deux processus sur V2,

- Elle permet de respecter l'ordre des jetons. Les deux processus de V2 s'exécutent en parallèle et sur des processeurs qui peuvent être différents. On ne peut pas prédire lequel des deux processus finira en premier (les processeurs sur lesquels ils s'exécutent peuvent être hétérogènes et donc de puissance différente). Deux cas de figure se présentent, soit le numéro un finit en premier (figure 4.16), soit le numéro deux termine le premier (figure 4.17). Dans le deuxième cas, il faut éviter que le jeton de valeur 16 soit utilisé par V3 avec le jeton de valeur 40. Les étiquettes de chaque jeton permettront au jeton de valeur 16 d'attendre l'arrivée du jeton de valeur 20,

- Elle permet de régler les problèmes de synchronisation entre les différentes répliques d'un groupe de redondance active. Dans la figure 4.18, nous avons représenté un groupe de noeuds de redondance : les noeuds Vi. Le noeud N2 s'attend à recevoir trois entrées (de V1, V2 et V3). Supposons que V1 tombe en panne, N2 effectue le vote sur les entrées de V2 et V3. La numérotation des jetons permet de traiter le cas ou V1, après quelques instants de silence recommence à émettre

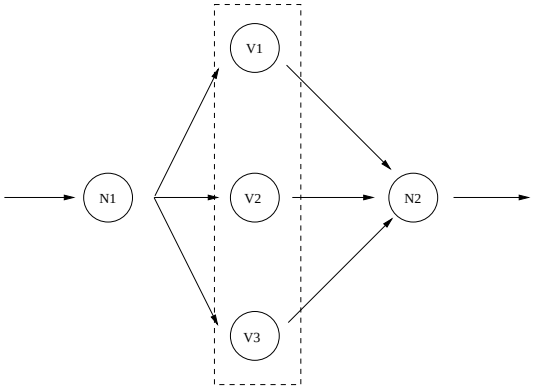

Figure 4.18: Le vote dans COSMOS

des jetons : l'émission d'un jeton étiqueté par un vieux numéro est alors ignoré par N2,

- Elle permet la gestion des points de reprise. Chaque jeton est enregistré sur un support stable **avec son étiquette**. Les numéros des étiquettes en cours de traitement sur chaque noeud sont aussi écrits sur disque. En cas de panne d'un noeud, les résultats des processus en cours de calcul sont perdus : on redémarre alors toutes les instances du noeud dont les entrées sont présentes. **C'est le point fort de COSMOS : on ne stocke pas l'espace mémoire et la pile des processus mais uniquement les résultats qui sont représentés par les jetons. Les journaux sont donc de faible volume et c'est certainement une des raisons pour laquelle COSMOS peut utiliser des points de reprise.**

4.5 Conclusion

Nous avons donné la description de plusieurs systèmes distribués offrant aux utilisateurs des outils de tolérance aux pannes. Il en existe bien sûr beaucoup d'autres mais les exemples ci-dessus permettent de donner une idée des différentes technologies utilisées aujourd'hui. Ces outils sont implantés, soit directement dans le système (c'est le cas de Delta 4, COSMOS ou GATOSTAR), soit sous forme de bibliothèques (comme Isis), soit grâce aux fonctionnalités du langage de programmation (par héritage ou réflexivité). Certains permettent une utilisation transparente de la tolérance aux pannes (Delta 4, GATOSTAR), d'autres obligent le programmeur à mélanger son programme

avec les directives du système (pour spécifier à quel moment les points de reprise doivent être effectués). Dans le cadre d'applications temps réel, la redondance active est presque toujours utilisée. Les points de reprise sont plus souvent associés à des applications moins contraintes dans le temps. Il existe toutefois quelques exemples où ceux-ci sont choisis pour des applications temps réel (comme COSMOS). **Nos expérimentations utiliseront donc des techniques de redondance active et de points de reprise.** Nous pouvons maintenant décrire dans le chapitre suivant le sous-système Saturne sur CHORUS/COOL.

Chapitre 5

Un sous-système Saturne sur CHORUS/COOL tolérant les pannes

5.1 Introduction

Dans ce chapitre, nous allons décrire les techniques nécessaires pour implanter le modèle Saturne avec des propriétés de tolérance aux pannes sur CHORUS/COOL. Les objectifs de cette étude sont de déterminer dans quelles conditions Saturne peut être réalisé sur un système d'exploitation temps réel à l'aide d'un bus à objets au standard CORBA, en ajoutant des propriétés de tolérance aux pannes. Nous n'aborderons pas CHORUS, COOL et CORBA qui ont déjà fait l'objet d'une description dans [21]. Nous nous intéressons plus spécialement dans ce document aux techniques de tolérance aux pannes que nous pouvons associer avec Saturne. La version du modèle Saturne utilisée pour nos expérimentations est la version mono-synchrone. Bien que ce modèle existe maintenant depuis plusieurs années, aucune implantation n'a été réalisée, à notre connaissance, **dans un environnement temps réel distribué**, alors que son objectif est l'exécution d'applications temps réel. De même, l'étude des mécanismes de tolérance aux pannes ne semble pas être un problème qui ait été abordé. L'originalité de notre implantation est donc l'utilisation d'une plate-forme temps réel pour la mise en oeuvre de Saturne associé à des mécanismes de tolérance aux pannes.

Ce chapitre est découpé en deux parties. La première partie décrit le sous-système Saturne et les différents mécanismes de tolérance aux pannes que nous souhaitons utiliser. La description du sous-système Saturne donnée ne comporte que les informations nécessaires au lecteur pour comprendre les

mécanismes de tolérance aux pannes que nous avons choisis. Une description plus détaillée peut être consultée dans [21]. La deuxième partie cite les prototypes réalisés et donne un bilan sur les résultats obtenus.

5.2 Conception du sous-système Saturne sur CHORUS/COOL

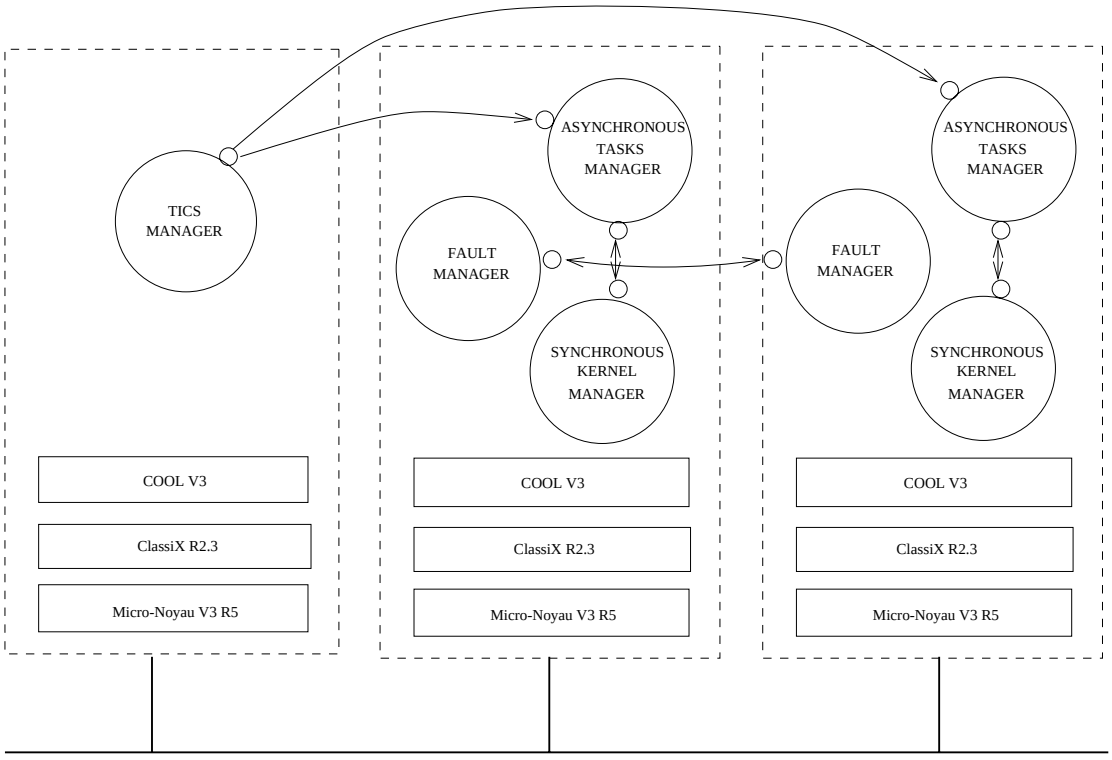

Figure 5.1: Architecture du sous-système Saturne

5.2.1 Architecture du sous-système Saturne

Dans le modèle Saturne, les grappes sont connectées par deux supports de communication : un support synchrone, un autre asynchrone. Dans nos simulations, nous n'implanterons pas l'utilisation du support asynchrone, c'est à dire les communications entre tâches transformationnelles. Nous simulerons uniquement les communications synchrones entre les noyaux. Nous ne faisons aucune supposition quant à la répartition des grappes sur le réseau :

on peut imaginer qu'il y ait une ou plusieurs grappes par processeur. Ces grappes se composent de deux parties:

- Le noyau réactif,
- Les tâches transformationnelles.

Les abstractions de CHORUS s'adaptent facilement à cette architecture. Les composantes de Saturne sont réalisées sous forme d'acteurs CHORUS qui sont :

- L'AM (Asynchronous tasks Manager) : cet acteur contient toutes les tâches transformationnelles, l'interface réactive de Saturne et un ordonnanceur pour les tâches transformationnelles. Chaque tâche transformationnelle est exécutée par une activité CHORUS. L'interface réactive et l'ordonnanceur sont implantés de manière identique. L'interface réactive est bloquée en attente de messages venant du noyau synchrone. Les communications entre les noyaux synchrones et les interfaces réactives sont décrites dans le paragraphe suivant. L'ordonnanceur est décrit dans [21],
- Le SM (Synchronous kernel Manager) : cet acteur encapsule le code Esterel. Il représente un noyau synchrone du modèle Saturne. Il reçoit régulièrement des tops en provenance du TM. A chaque top reçu, le SM envoie ses commandes vers l'AM et diffuse ses signaux vers les autres noyaux synchrones,
- Le TM (Tics Manager) : c'est l'acteur qui génère à intervalles réguliers des tops en direction des SM et des FM. Il implante l'horloge globale du modèle synchrone faible. Le fonctionnement de cet acteur est décrit dans [21]. Il y a un seul TM dans un système CHORUS,
- Le FM (Fault Manager) : cet acteur est responsable de la mise en oeuvre des mécanismes de tolérance aux fautes que nous souhaitons utiliser. Son fonctionnement est décrit dans le paragraphe quatre ainsi que dans la deuxième partie de ce chapitre. Il y a un FM par site CHORUS.

Pour des contraintes techniques et des soucis de performances, les SM et FM sont des acteurs superviseurs. Les AM sont des acteurs utilisateurs. L'architecture du sous-système CHORUS/COOL est représentée dans la figure 5.1.

5.2.2 Les noyaux synchrones

Dans ce paragraphe, nous allons décrire les acteurs SM, puis nous verrons le mode de communication utilisé entre les noyaux synchrones et entre chaque noyau synchrone et leur interface réactive.

* **Architecture des acteurs SM**

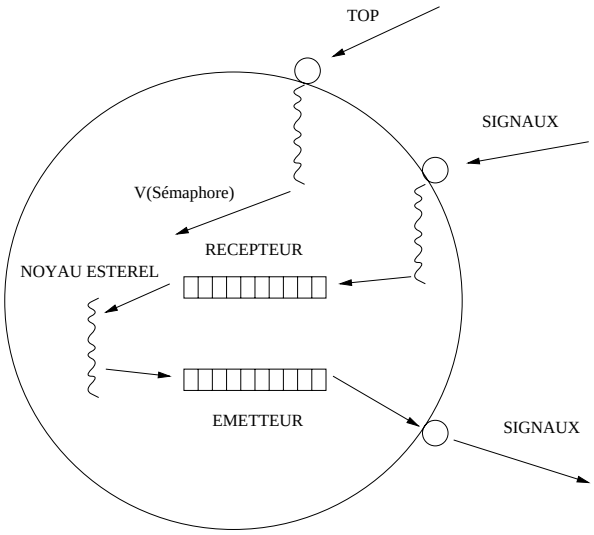

Figure 5.2: Description de l'acteur SM

Les acteurs SM, comme le montre la figure 5.2, sont composés de :

1. Trois activités :
   - Une première activité qui exécute le noyau Esterel et qui est bloquée sur un sémaphore en attente des signaux à consommer dans la file d'attente de réception,
   - Une deuxième activité chargée de la réception des signaux. Elle reçoit et stocke dans la file de réception les signaux en continu,
   - Une troisième activité qui attend les tops émis par le TM. A chaque top, elle détermine les signaux qui peuvent être délivrés au noyau Esterel grâce aux temps de latence en réception. Elle réveille ensuite le noyau Esterel,
2. Deux files d'attente qui permettent de stocker les signaux jusqu'à ce que les temps de latence soient écoulés,

3. Une horloge H_{tops} qui stocke l'horloge globale courante. Cette horloge est partagée par les trois activités de l'acteur. Avec cette horloge qui est une horloge logique, les activités déterminent si les temps de latence des signaux sont écoulés. Les temps de latence sont définis dans le chapitre trois.

* Communications entre les noyaux synchrones

Toutes les communications entre noyaux sont exécutées par des messages asynchrones. Bien que les messages soient envoyés de manière asynchrone, leur transport doit être fiable. Les communications asynchrones permettent d'augmenter le parallélisme des réceptions et des émissions de requêtes et donc de diminuer l'intervalle entre deux tops d'horloge. Les temps de latence sont simulés par un retard en nombre de top. Le fonctionnement du SM pendant un cycle peut se résumer de cette façon :

- L'activité de réception reçoit tous les signaux envoyés par les autres noyaux. Cette réception est continue. Chaque signal reçu est stocké dans la file de réception et est estampillé par l'horloge H_{tops},
- L'activité en attente des tops est réveillée par le TM. A cet instant, elle met à jour H_{tops}. Puis, elle regarde tous les signaux de la file de réception qui peuvent être délivrés (en fonction du temps de latence L_{recep} associé à chaque signal). Enfin, elle débloque le noyau Esterel, puis, se met en attente du prochain top,
- L'activité du noyau Esterel est bloquée sur son sémaphore. Quand elle est débloquée, elle consomme les signaux que l'activité en attente des tops lui a donnés. Elle effectue ensuite une transition de l'automate Esterel. Les signaux de sortie qu'elle émet pendant la transition sont estampillés par H_{tops} et sont stockés dans la file d'attente en émission. A la fin de son exécution, elle consulte la file d'attente en émission pour voir si elle peut envoyer des signaux. Elle tient compte de l'estampille et du temps de latence associé à chaque signal,
- Puis on recommence un autre cycle au prochain top du TM.

* Protocole entre l'AM et le noyau synchrone

Les interactions entre un noyau synchrone et son interface réactive sont de trois types :

- Le noyau synchrone peut créer, suspendre, réactiver ou détruire une tâche. Ces commandes sont synchrones et ne fournissent pas de résultat en retour,

- Le noyau peut aussi obtenir des informations sur les tâches. Si l'interface réactive et les tâches transformationnelles sont placées dans le même acteur c'est pour faciliter la réalisation de ce type de commande. En effet, les tâches et l'interface réactive partageant le même espace d'adressage, l'interface réactive peut directement lire les données mises à jour par les tâches transformationnelles. C'est ainsi que l'on réalise les commandes Saturne CONSULT et KILL. Un certain nombre de commandes non Saturne que nous avons mises en place et qui permettent, entre autre, la consultation de l'état d'une tâche, sont réalisées grâce à la même technique,

- Enfin, la dernière interaction entre l'AM et le noyau concerne la terminaison des tâches transformationnelles. Quand une tâche transformationnelle se termine, celle-ci met à jour son état. Lorsque le noyau synchrone a terminé d'envoyer ses commandes à l'interface réactive, il lui demande si certaines tâches sont terminées. L'interface réactive consulte alors les tâches, puis renvoie les numéros des tâches terminées.

5.2.3 Implantation du modèle Saturne sur COOL

*** Les objets COOL de Saturne**

Jusqu'à présent, nous avons décrit la structure du sous-système Saturne sans pour autant décrire la manière dont nous utilisions COOL pour la mettre en oeuvre. C'est l'objet de ce paragraphe.

Une application COOL est constituée d'objets serveurs et d'objets clients. Nous modélisons donc le sous-système Saturne comme un ensemble d'objets. Dans Saturne, on trouve deux types d'objets COOL :

- Les objets de type "interface réactive". Dans chaque AM on trouve un objet de ce type. Il permet au noyau synchrone d'exécuter les commandes Saturne sur les tâches transformationnelles. Un exemple de code IDL est donné dans la figure 5.3. On y retrouve toutes les commandes Saturne définies dans le chapitre trois,

```
♯include <COMPLEX.idl>
interface callIR {
        void start-T1(out long identificateur, in long p1, in long p2);
        void start-T2(out long identificateur, in COMPLEX p1,
                in COMPLEX p2, in COMPLEX p3);
        void start-T3(out long identificateur,in string s1, in long p1);
        void consult-T1(in long identificateur,out long p1);
        void consult-T2(in long identificateur,out COMPLEX p1);
        void consult-T3(in long identificateur,out long p1);
        void kill(in long identificateur);
        void suspend(in long identificateur);
        void resume(in long identificateur);
        boolean isFinished(in long identificateur);
        boolean isSuspended(in long identificateur);
        boolean isActive(in long identificateur);
};
```

Figure 5.3: L'interface IDL d'un acteur AM

```
♯include "HAUTEUR.idl"
interface NoyauInput {
        oneway void receiveMessage-APPUI-HOMING();
        oneway void receiveMessage-CR-SCVSDT-PERTE();
        oneway void receiveMessage-MODIFICATION-HS(in HAUTEUR data);
        oneway void receiveMessage-SELECTION-FORCAGEMER-OFF();
        oneway void receiveMessage-SELECTION-FORCAGEMER-ON();
};
```

Figure 5.4: L'interface IDL d'un noyau synchrone

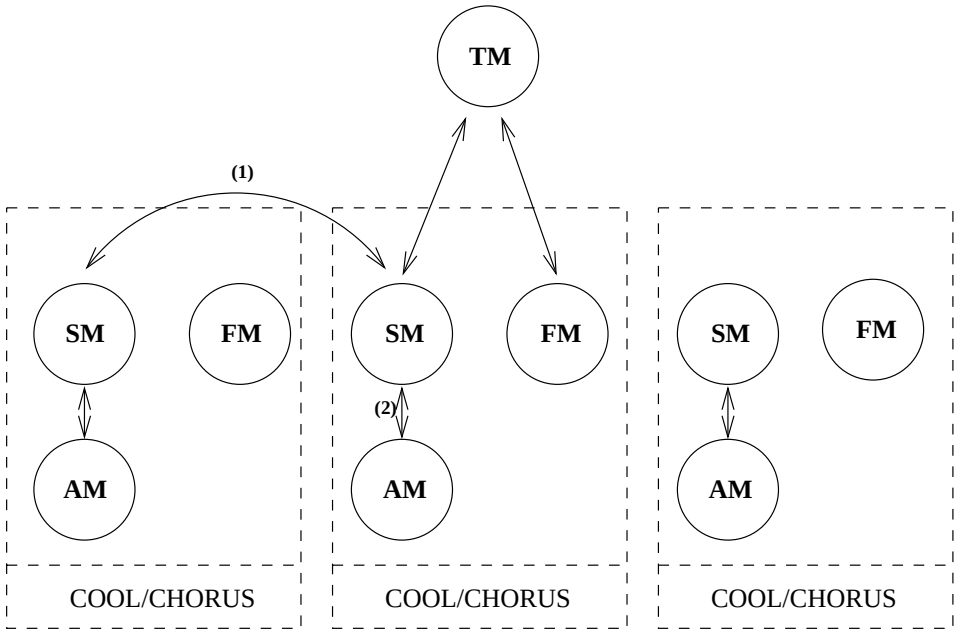

Figure 5.5: Communications du sous-système Saturne réalisées en COOL

- Les objets de type “objets synchrones”. Chaque objet de ce type encapsule un automate Esterel. Ces "objets synchrones" sont constitués des éléments que l'on a vu dans la figure 5.2 : c'est à dire des files de réception et d'émission. Chaque objet synchrone exporte une interface IDL qui comprend tous les signaux que les autres noyaux peuvent lui envoyer. En fait, chaque méthode IDL correspond au dépôt d'un signal. L'invocation d'une de ces méthodes place un message dans la file de réception du noyau destinataire. Un noyau *a* qui souhaite envoyer un signal *SIG* au noyau *b* invoquera la méthode *receiveMessage-SIG*() du noyau *b*. La figure 5.4 nous donne un exemple d'IDL d'un objet synchrone.

*** Modes de communication utilisés pour les invocations de méthodes**

La figure 5.5 représente l'architecture Saturne. Dans cette architecture, seules les communications issues de l'applicatif sont faites en COOL. Toutes les communications concernant les acteurs systèmes du sous-système Saturne sont développées en IPC[1] CHORUS. Les flèches (1) et (2) qui représentent

[1]IPC pour **I**nter **P**rocess **C**ommunication.

les communications entre applicatifs effectuées grâce à COOL sont :

1. Les échanges de signaux Esterel entre les noyaux synchrones : c'est une communication point à point. Elle est effectuée de manière asynchrone. On utilise un appel de méthode avec l'attribut IDL "*oneway*". L'appel de méthode "*oneway*" est essentiellement justifié pour augmenter le parallélisme entre les différentes communications,

2. Les échanges entre interface réactive et noyau synchrone: c'est une communication point à point synchrone. On utilise donc un appel de méthode CORBA synchrone. L'appel synchrone à l'interface réactive est justifié par le fait que l'exécution des commandes Saturne doit être faite dans l'instant d'exécution du noyau synchrone. De plus, certaines méthodes renvoient des résultats (c'est le cas des commandes Saturne CONSULT).

5.2.4 La tolérance aux pannes

*** Objectifs des mécanismes proposés**

L'existence de fonctions critiques dans les applications qui devront fonctionner avec Saturne nécessite l'utilisation de mécanismes de tolérance aux pannes. Dans un système Saturne, on peut classer les fautes selon trois origines :

- Les fautes dues aux pannes d'une machine,
- Les fautes dues aux pannes d'un noyau synchrone,
- Les fautes dues aux pannes de tâches transformationnelles.

Les tâches transformationnelles ne sont pas prouvées formellement, il est donc nécessaire de prendre en compte leurs éventuelles anomalies de fonctionnement. Toutefois, il est difficile de déterminer les traitements à réaliser pour recouvrir une faute d'une tâche transformationnelle. En effet, les mécanismes à mettre en oeuvre sont dépendants de la tâche elle-même. L'environnement d'exécution doit donc se contenter de fournir un cadre générique pour permettre à l'applicatif de définir ses mécanismes spécifiques. A ce titre, nous ne nous intéresserons pas ici aux tâches transformationnelles. Par contre, la panne d'un site ou d'un noyau synchrone n'empêche pas de définir des mécanismes standards de recouvrement qui fonctionnent quelque soit le code Esterel encapsulé. Les objectifs des mécanismes proposés sont donc de traiter les pannes franches de sites ou de noyaux synchrones ainsi que les pannes

temporelles des noyaux synchrones. Nous définissons un noyau synchrone provoquant une panne temporelle comme un noyau n'ayant pas terminé les traitements et/ou communications déclenchés par un top t alors que le top $t+1$ arrive.

Les systèmes temps réel que nous avons vus dans le chapitre quatre utilisaient comme mécanisme de tolérance aux pannes :

- Presque toujours la redondance active. (Nous avons présenté les exemples de Delta 4, MARS et COSMOS),
- Parfois les points de reprise. C'est le cas de COSMOS où le modèle flot de données est exploité afin d'obtenir des journaux de faible volume, et donc un temps de recouvrement accéléré. Dans cet exemple, tous les processus enregistraient leur point de reprise en même temps.

Nous nous proposons donc d'utiliser ces deux techniques pour recouvrir les pannes que nous avons décrites ci-dessus.

* **Utilisation de la redondance active**

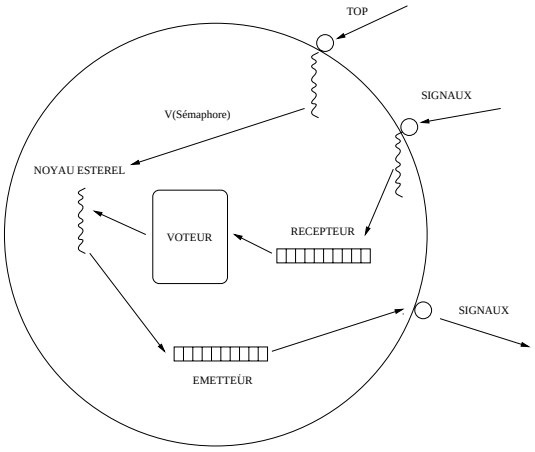

Figure 5.6: Mécanismes de vote sur Saturne

La redondance active nous permet de recouvrir **rapidement** des pannes de sites ou de noyaux synchrone. Ici, l'entité qui est répliquée est le noyau synchrone. Toutes les répliques d'un noyau synchrone sont rassemblées dans un groupe d'objets COOL. Le vote est effectué dans l'acteur SM (voir figure 5.6) à chaque réception d'un top horloge, en comparant tous les signaux envoyés par les membres d'un même groupe de répliques. La méthode de vote

qui détermine la valeur du signal considérée comme juste est accessible par l'utilisateur (l'utilisateur peut donc implanter un vote majoritaire ou tout autre politique comme celles décrites dans Electra). L'utilisation de redondance active avec Saturne est beaucoup plus simple que dans les systèmes où les communications sont asynchrones. En effet, Saturne fournit une horloge globale, de ce fait, nous n'avons pas besoin d'algorithmes de diffusion possédant des propriétés d'ordre. Cependant, ceci ne nous dispense pas de l'utilisation de diffusion atomique des signaux sur toutes les répliques du groupe. Nous avons vu dans Delta 4 que certaines hypothèses sur le comportement des programmes et des machines étaient formulées, et en particulier, la nécessité d'avoir des programmes déterministes lorsque l'on souhaitait utiliser la redondance active. Utiliser des noyaux Esterel comme unité de redondance permet de s'affranchir de cette contrainte : en effet, **le code Esterel a un comportement logique et temporel déterministe**. Il n'y a bien sûr aucune garantie de déterminisme pour le code écrit en langage hôte par le programmeur. Les mécanismes présentés dans ce paragraphe, qui sont implantés dans le premier prototype, ne tiennent pas compte du comportement des sites. Dans le cas de la redondance active, le fait que les sites soient "*fail-silent*" ou "*fail-uncontrolled*" n'a d'influence que sur le nombre de répliques nécessaires pour détecter t pannes simultanées et non sur les mécanismes eux mêmes (voir chapitre quatre, paragraphe sur Delta 4).

*** Utilisation de points de reprise**

Après l'utilisation de la redondance active pour augmenter le niveau de fiabilité d'une application Saturne, nous avons tenté d'y associer des points de reprise. Cette association permet de maintenir un nombre identique de répliques dans un groupe même en cas de défaillance. L'utilisation de points de reprise dans les systèmes temps réel est moins courante car elle ne permet pas un recouvrement des fautes aussi rapide que la redondance active. Toutefois, elle peut être envisagée pour des fonctionnalités peu critiques du système, ou, comme dans COSMOS, quand des mécanismes accélérant le recouvrement de la panne peuvent être utilisés. Afin de diminuer le temps de recouvrement d'un noyau synchrone et de diminuer l'impact sur les performances de l'enregistrement des points de reprise, nous nous appuyons sur les deux observations suivantes :

- Faire exécuter une transition à un automate Esterel est une tâche qui est rapide et qui demande peu de ressource processeur (si l'on ne tient pas compte de l'émission des signaux de sortie de l'automate qui sont réalisées dans nos prototypes par des invocations "*oneway*" de méthode COOL),

- Les données véhiculées par les signaux Esterel semblent être de faible volume dans les applications d'avioniques. Dans [21], nous avons implanté une application Saturne de suivi de terrain comportant 140 signaux Esterel et dont :
  - 54,24 % sont des signaux purs,
  - 17,80 % sont des signaux véhiculant des entiers courts,
  - 16,94 % sont des signaux véhiculant des booléens,
  - 11,02 % sont des signaux véhiculant des entiers longs.

Nous constituons nos points de reprise par l'enregistrement des signaux d'entrée délivrés à l'automate lors de chaque top. Lorsque l'on souhaite relancer un noyau synchrone, on lit le journal et on délivre les signaux d'entrée reçus en activant l'automate Esterel. Durant cette phase, les signaux en sortie ne sont pas réémis. Ces points de reprise sont donc particuliers puisqu'ils ne contiennent pas l'état de l'automate Esterel mais les informations qui permettent de le reconstituer rapidement. Il est possible, lors de la compilation, d'isoler les variables d'états de l'automate. On pourrait donc utiliser une technique identique à celle de GATOSTAR, en associant l'enregistrement régulier d'états de l'automate avec une journalisation des signaux reçus. On écrirait les signaux sur le disque afin de reconstituer à partir du dernier état enregistré l'état précédent la panne. Cette technique, contrairement à GATOSTAR, ne serait pas justifiée pour obtenir un état global cohérent, puisque l'horloge globale de Saturne nous aide déjà à définir cet état, mais pour accélérer encore le recouvrement d'une panne. Toutefois, ceci demande une analyse de la compilation de programmes Esterel qu'il n'était pas possible de réaliser dans le cadre de cette étude. Dans tous les cas, l'enregistrement des points de reprise est facilité par Saturne grâce à son horloge globale qui synchronise tous les noyaux.

Avant de pouvoir relancer un noyau synchrone, il faut détecter sa panne. Nous utilisons une détection passive et active. La détection passive est réalisée par les noyaux synchrones. Les noyaux synchrones constituant l'application Saturne détecte les pannes lors du vote. Si lors d'un vote, ils constatent qu'ils leur manquent un signal (ils connaissent le contenu des groupes de redondance), ils savent qu'un noyau est victime, soit d'une panne franche, soit d'une panne temporelle. La détection passive peut entraîner une latence importante entre l'occurrence de la panne et la détection de la panne. En effet, un noyau ne reçoit pas systématiquement à chaque top un signal d'un autre noyau. Il peut recevoir un signal tous les n tops. De plus, avec l'utilisation

de la détection passive seule, on ne détecte pas les pannes des noyaux n'émettant aucun signal. Aussi, nous utilisons un mécanisme de détection active qui consiste à rajouter des noyaux Esterel dont le seul but est de recevoir des signaux des noyaux de l'application Saturne. Ces noyaux de détection permettent de régler, si nécessaire, les deux problèmes ci-dessus.

Une fois la panne détectée, l'acteur FM se charge de la recouvrir. Dans nos prototypes, il existe un acteur FM par site. Ils sont tous rassemblés dans un groupe d'objets COOL (chaque FM exporte une interface IDL décrite dans la partie suivante). On définit la notion de mode de fonctionnement. A chaque mode est associé une répartition différente des noyaux Esterel. En cas de panne d'un noyau seul, l'application reste dans le mode de fonctionnement courant. Lors de la panne d'un site, elle bascule dans un mode où il est prévu qu'aucun acteur ne s'exécute sur le site en panne. Lorsqu'un noyau synchrone avertit un FM de l'occurrence d'une panne, il avertit tous les FM car il effectue une diffusion sur groupe. Tous les FM effectuent ensemble, si nécessaire, le basculement dans le nouveau mode de fonctionnement. Chaque FM gère sur son site la création et la destruction de noyaux. La destruction d'un noyau intervient lors de la détection de sa panne. En effet, un noyau a peut avertir un FM de la panne d'un noyau b alors que le noyau b existe toujours. C'est notamment le cas pour une panne temporelle. La création de noyau a lieu lors d'un basculement de mode, où tout simplement après détection de la panne d'un noyau seul. La détection de la panne d'un site est réalisée par les FM qui considèrent un site en panne lorsque tous les noyaux du site de répondent plus. Les noyaux relancés s'exécutent hors du groupe tant qu'ils n'ont pas rattrapé l'état courant des autres répliques. Une fois complètement réinitialisés, ils réintègrent le groupe de vote. La panne est alors totalement recouverte. Les mécanismes décrits ci-dessus ont été en partie implantés dans le prototype deux de ce document.

Nous avons décrit l'architecture du sous-système Saturne sur CHORUS-COOL ainsi que les mécanismes de tolérances aux pannes que nous souhaitions expérimenter. La partie suivante de ce chapitre décrira les deux prototypes développés et dressera un rapide bilan.

5.3 Description et bilan des prototypes réalisés

Dans cette partie, nous allons décrire successivement les deux prototypes réalisés pour tester les mécanismes de tolérances aux pannes. Nous commencerons par décrire le prototype utilisant une redondance active, puis, celui associant de la redondance active avec des points de reprise. L'application

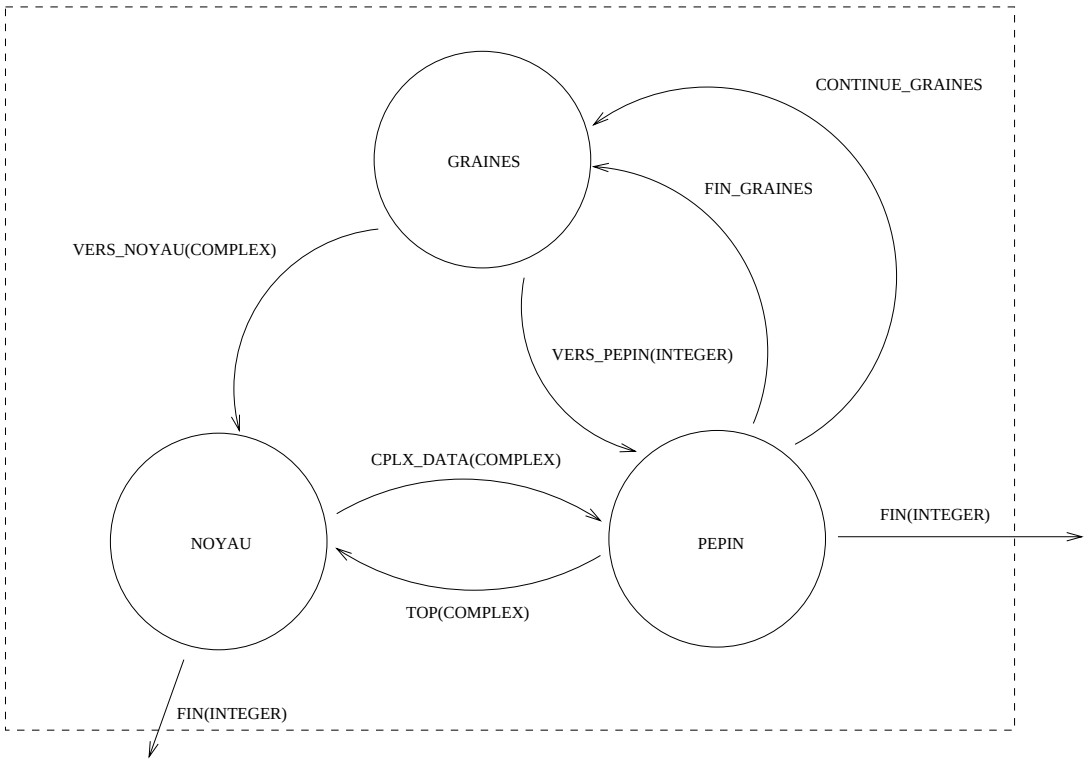

Figure 5.7: Le premier prototype

Esterel utilisée dans les deux cas fera l'objet du premier paragraphe. Il faut noter que cette étude a abouti sur la réalisation de quatre prototypes, mais seuls les deux décrits ci-dessous intègrent des mécanismes de tolérance aux pannes. Les deux autres sont présentés dans [21].

5.3.1 L'application Esterel utilisée

L'application Esterel utilisée est simple. Elle est constituée de trois noyaux : *Pepin*, *Noyau* et *Graines*. Les noyaux ainsi que les signaux qu'ils s'échangent sont représentés dans la figure 5.7. Chaque cercle correspond à un noyau synchrone. Les arcs représentent les signaux. Ils portent un nom et sont orientés. L'orientation des arcs correspond aux sens des communications entre les noyaux. Si un signal transporte une valeur, le type de cette valeur est spécifié entre parenthèses après le nom du signal. Le cadre en pointillé délimite l'application Saturne et l'environnement extérieur.

* **Le noyau *Pepin***

Ce noyau manipule un type utilisateur qui est la représentation des nombres complexes. Une seule tâche transformationnelle est contrôlée par ce noyau synchrone.

*** Le noyau *Noyau***

Ce noyau est plus complexe que le premier. Il contrôle trois tâches transformationnelles. De plus, il est constitué de six tâches Esterel s'exécutant en parallèle. Il manipule aussi des types complexes.

*** Le noyau *Graines***

Ce dernier noyau est le plus simple des trois. Il ne gère aucune tâche transformationnelle.

*** Description du fonctionnement de l'application Esterel**

Les deux noyaux synchrones *Noyau* et *Pepin* s'échangent des signaux constitués d'un nombre complexe. Ces signaux sont TOP et CPLX_DATA. Ils affichent tous les deux la fin de leurs tâches transformationnelles grâce à l'envoi d'un signal FIN vers l'environnement extérieur. Ce signal FIN comporte l'identifiant de la tâche terminée. *Noyau* démarre quatre tâches dès le début de son exécution (deux tâches T1 et une tâche T2 et T3). A la terminaison de sa tâche T2, il envoie le résultat de calcul de cette tâche vers *Pepin* grâce au signal CPLX_DATA. *Pepin* est inactif jusqu'à la réception de ce signal. Lorsqu'il reçoit le signal CPLX_DATA, il démarre lui aussi une tâche T2, attend que celle-ci se termine, puis renvoie le résultat à *Noyau*. *Graines*, quant à lui, diffuse à chaque top un entier vers *Pepin* et un nombre complexe vers *Noyau*. Il cesse ses émissions quand *Pepin* reçoit le signal CPLX_DATA.

5.3.2 Le premier prototype : utilisation de la redondance active

La redondance active est implanté par un vote sur la réception des signaux Esterel. Les groupes de signaux sont réalisés à l'aide de groupe d'objets COOL. La figure numéro 5.8 décrit l'application Esterel et les groupes d'objets COOL utilisés. Le noyau *Pepin* est tripliqué et le noyau *Graines* est dupliqué. Ici, on applique un double vote sur les signaux Esterel. L'émission d'un signal vers un groupe de noyaux Esterel est réalisée par un appel de méthode sur un groupe d'objets COOL. Si le noyau émetteur fait lui même partie d'un groupe de n_e objets, chaque noyau récepteur reçoit alors

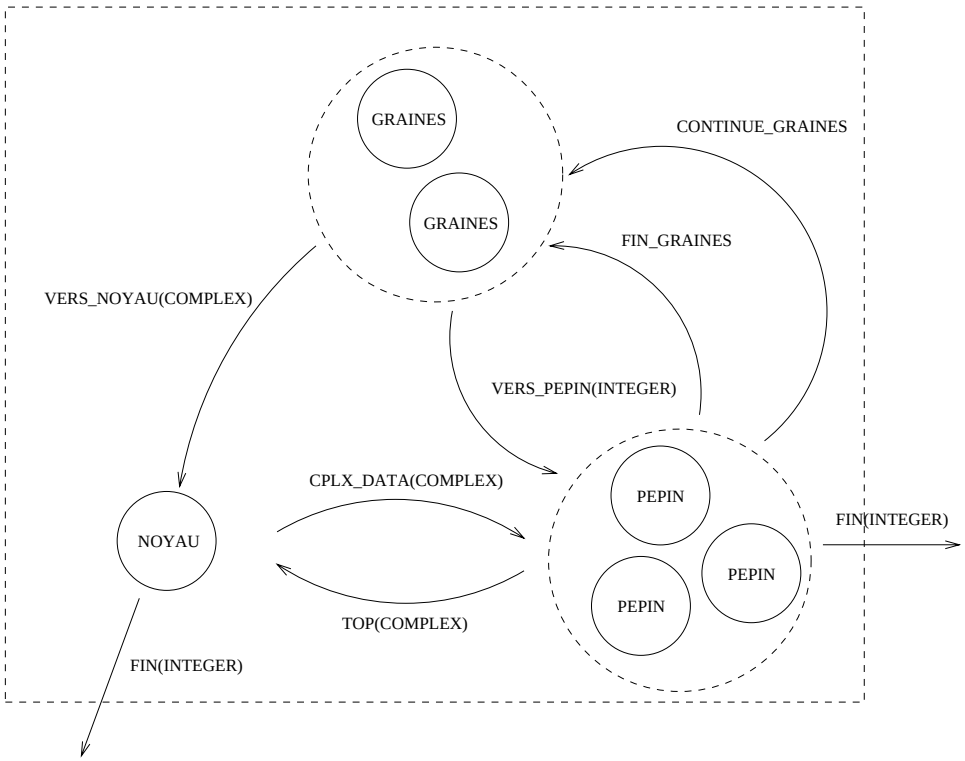

Figure 5.8: L'application Esterel utilisée pour la mise en oeuvre du vote

n_e messages. Le premier vote est donc effectué sur ces n_e messages reçus. Le deuxième vote est aussi réalisé par les noyaux récepteurs. Quand les noyaux récepteurs ont effectués le vote sur leurs n_e messages, ils s'échangent le message qu'ils considèrent comme juste et effectuent un deuxième vote. Si le groupe des récepteurs contient n_r noyaux synchrones, alors ils reçoivent chacun $n_r - 1$ messages sur lesquels ils effectueront un vote. Ceci constitue ce que nous avons appelé le deuxième niveau de vote. La figure 5.9 illustre ces échanges de messages : les flèches (1) correspondent aux messages utilisés pour le premier vote, les flèches (2) pour le deuxième vote. Les noyaux émetteurs sont les *Pepins*, les récepteurs sont les *Graines*. Ce mécanisme de double vote constitue un double contrôle qui permet :

- De détecter la panne d'un noyau émetteur ou récepteur plus rapidement,
- De garantir que les noyaux récepteurs calculeront leur transition avec la même valeur du signal d'entrée.

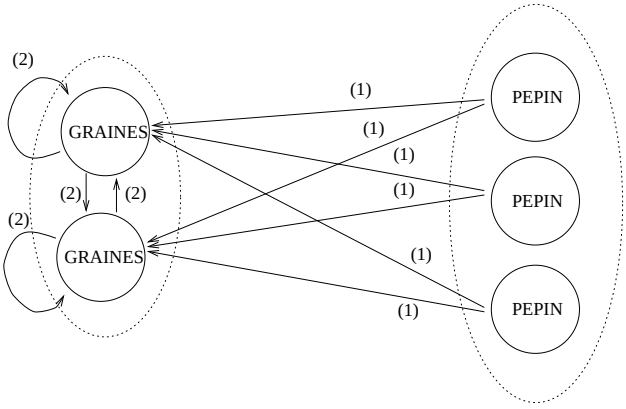

Figure 5.9: Le mécanisme de double vote

Ce mécanisme de redondance active est totalement transparent pour les noyaux qui émettent les signaux. En effet, un noyau qui effectue une émission invoque une méthode dont il ne sait pas si elle fait partie d'un seul objet ou d'un groupe d'objets COOL.

Les problèmes auxquels nous avons été confrontés pour ce prototype concernent l'invocation sur groupe de COOL. La version de COOL que nous avons utilisée (COOL V3.R0b) n'est pas adaptée à la tolérance aux pannes. Dans cette version, il est impossible de stopper brutalement un objet sans entraîner une panne du groupManager ou un blocage des objets qui étaient en communication avec l'objet fautif. Dans le cadre de nos expérimentations, nous avons simulé la panne par un arrêt normal de l'acteur : COOL permet l'ajout ou la suppression dynamique de serveurs dans un groupe d'objets. Ajoutons que COOL ne permet pas aux utilisateurs de spécifier de nouvelles politiques d'invocation. Dans notre cas, un protocole de diffusion atomique nous aurait été nécessaire. Enfin, il est actuellement impossible d'utiliser des invocations *"oneway"* sur des groupes d'objets.

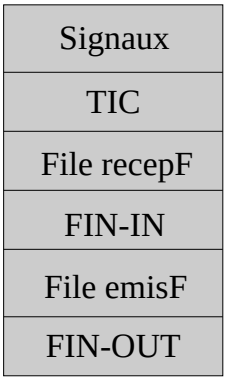

Figure 5.10: Le journal des automates Esterel

```
application proto1:

        kernel pepin 3 SM1;
        kernel noyau 1 SM2;
        kernel graines 2 SM3;

        instance pepin interPepin1;
        instance pepin interPepin2;
        instance pepin interPepin3;
        instance noyau interNoyau;
        instance graines interg1;
        instance graines interg2;

        mode pepin 1 m0 2116;
        mode noyau 1 m0 2116;
        mode graines 1 m0 2116;
        mode graines 1 m1 2116;
        ⋮
        mode graines 2 m1 2116;

        log pepin 1 p1log;
        log noyau 1 n1log;
        log graines 1 g1log;
 end;
```

Figure 5.11: Exemple d'un fichier de configuration

5.3.3 Le deuxième prototype: association de la redondance active avec des points de reprise

Le deuxième prototype utilise des points de reprise pour maintenir un nombre identique de répliques dans chaque groupe de vote. Ces points de reprise sont particuliers dans le sens où ils ne contiennent pas l'état de l'automate Esterel mais les informations permettant de le reconstituer (voir figure 5.10). Les noyaux chargés d'enregistrer les points de reprise effectuent une écriture à chaque top d'horloge de Saturne. Chaque écriture est constituée de:

- La liste des signaux envoyés à l'automate Esterel durant le top courant. Chaque signal est codifié par un entier. On stocke sur fichier le code correspondant au signal ainsi que sa valeur,
- Le contenu des files d'attente d'émission et de réception du SM encapsulant l'automate Esterel. En effet, s'il existe des temps de latence différents de zéro, ces files ne sont pas nécessairement vides.

Le volume de ces journaux dépend essentiellement de la durée d'exécution de l'application (nombre de top), du nombre de signaux échangés et du type des signaux. A titre d'information, les tailles moyennes des journaux de notre prototype, où les noyaux échangent 37,5 % de nombres complexes, 37,5 % d'entiers et 25 % de signaux purs, pendant 20 tops en moyenne sont de:

- 549 octets pour le noyau *Noyau* (il ne reçoit que des nombres complexes),
- 312 octets pour le noyau *Pepin* (il reçoit un entier et un nombre complexe),
- 280 octets pour le noyau *Graines* (il ne reçoit que des signaux purs).

Les concepteurs d'applications Saturne définissent un fichier de configuration qui stipule comment est utilisée la redondance active et les points de reprise. La figure 5.11 donne un exemple de ces fichiers. Ici, on y stipule le nombre de répliques pour chacun des noyaux *Pepin, Noyau* et *Graines*. On déclare aussi les modes de fonctionnement. Ces modes définissent sur quelles machines s'exécutent les différentes répliques d'un noyau synchrone. L'occurrence de pannes ou la remise en état d'un site fait passer l'application d'un mode à un autre. Afin de pouvoir disposer de suffisamment de ressource pour relancer les noyaux, on spécifie dans le mode de fonctionnement initial des sites qui ne sont pas utilisés. Sur ces sites, seul un FM fonctionne tant que le système n'a pas été victime de panne de sites. Enfin, le fichier de

configuration spécifie avec le mot clef *log* quels sont les noyaux synchrones qui effectuent les journalisations sur fichier des signaux. Le contenu de ce fichier de configuration est connu par tous les acteurs FM et SM du système.

```
interface fault {
        void faultDetection(in long kern, in long inst,in long siteSrc);
        void seeCourantMode(out long mode);
        void restartNode(in long numSite);
        void kernelIsReadyNow(in long kern, in long inst,in long siteSrc);
};
```

Figure 5.12: L'interface IDL du FM

Les problèmes auxquels nous avons été confrontés durant la mise en oeuvre de ce prototype sont :

- La réalisation de la détection des pannes. En effet, il nous a été nécessaire de numéroter toutes les répliques du système afin de les désigner aisément. Cette désignation est connue par tous les SM et FM puisqu'elle est issue du fichier de configuration que nous avons décrit précédemment. La figure 5.12 nous donne l'interface IDL des acteurs FM. Cette interface est utilisée par les SM détectant une panne qui invoquent alors la méthode *faultDetection*(). La numérotation permet au SM de désigner précisément quel est le noyau en panne. Cette interface est aussi invoquée par les FM. *seeCourantNode*() et *restartNode*() sont utilisés par un FM s'exécutant sur un site en cours de réinitialisation. Enfin la méthode *kernelIsReadyNow*() permet d'avertir les FM que la panne d'un noyau est recouverte et que celui-ci est de nouveau réinséré dans son groupe,

- L'insertion d'un noyau en cours de réinitialisation dans son groupe de vote. En effet, il faut synchroniser les noyaux du groupe avec les noyaux qui cherchent à se réinsérer dans leur groupe. La figure 5.13 représente un chronogramme de cette synchronisation. Pour résoudre ce problème, nous avons découpé un top Saturne en trois sous-tops. Les automates Esterel sont activés tous les trois sous-tops. Les noyaux fonctionnant normalement activent leur automate Esterel lors d'un top. Les calculs et les communications des automates durent alors un sous-top maximum. Sur la figure 5.13, l'activation des automates Esterel est

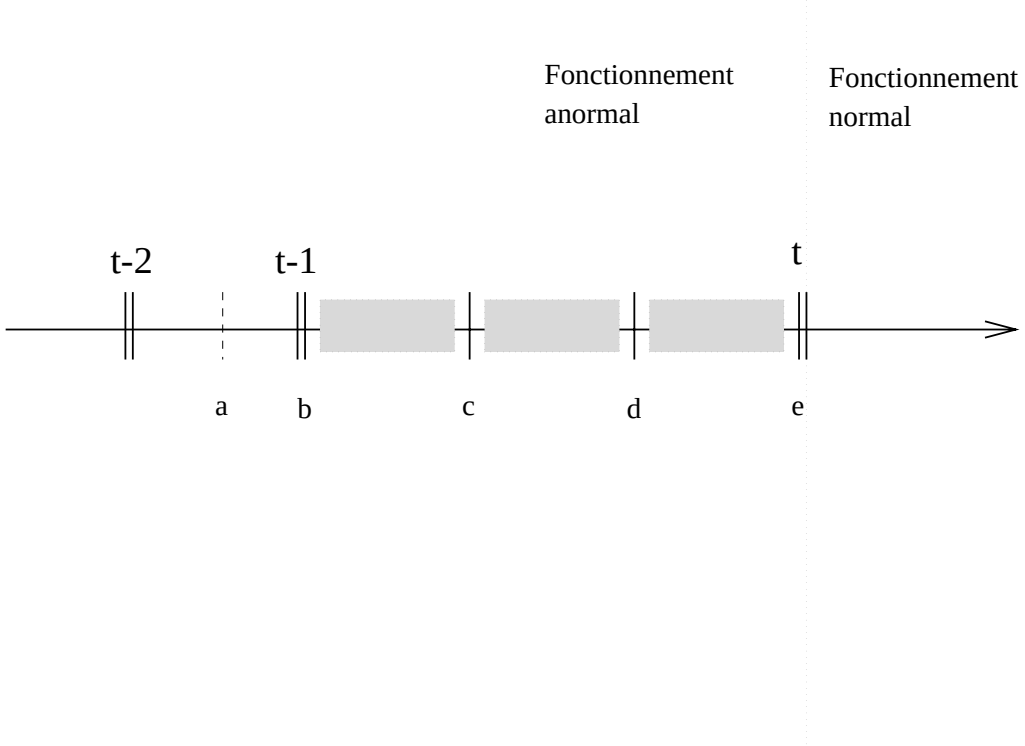

Figure 5.13: Chronogramme d'activation des noyaux

réalisé sur les points $t-2$, $t-1$ et t. Quand les automates sont activés au point b, leurs calculs et leurs communications sont terminés au point c. L'intervalle entre les sous-tops c et d est dédié à l'enregistrement des points de reprise. Tous les noyaux effectuent l'écriture en même temps et lorsque le sous-top d arrive, on sait que tous les noyaux ont terminés d'écrire leur point de reprise. Enfin, l'intervalle entre les sous-tops d et e permet aux noyaux en cours de réinitialisation de se réinsérer dans leur groupe de vote. Quand un noyau commence sa réinitialisation, il lit le journal, et restaure l'état de son automate Esterel. Lorsqu'il arrive à la fin du fichier, un top est en cours de réalisation et il doit attendre que celui-ci se termine. Sur notre chronogramme, la fin de fichier intervient au point a. Le noyau attend alors jusqu'au point b, puis, lit le reste du fichier journal tout en restaurant l'état de son automate au top d, enfin, durant l'intervalle d et e il se réinsère dans son groupe COOL.

5.4 Conclusion

Nous venons de présenter les mécanismes de tolérance aux pannes que nous avons expérimentés sur Saturne. Le premier prototype a été entièrement réalisé. Faute de temps, le deuxième n'a pu être complètement achevé. Si nous avons pu montrer qu'il était possible de détecter, relancer et réinsérer dans son groupe un noyau synchrone, les mécanismes de tolérance aux pannes sur les sites n'ont pas été totalement réalisés. Nous n'avons pas pu tester le cas où un site se réinitialise et redevient disponible pour l'application. Des tests supplémentaires sont donc nécessaires pour valider convenablement les mécanismes présentés dans ce chapitre.

Chapitre 6

Conclusion

Les objectifs de cette étude étaient de voir dans quelles conditions il était envisageable de mettre en oeuvre le modèle Saturne au dessus de CHORUS et de COOL. Ajouté à cet objectif principal, deux autres aspects devaient être analysés : la possibilité d'ajouter au modèle Saturne des propriétés de tolérance aux pannes et les apports des technologies CORBA pour le développement d'applications temps réel.

Ce document s'intéresse plus particulièrement aux objectifs concernant la tolérance aux pannes. Nous avons décrit dans le chapitre quatre quelques exemples de systèmes temps réel implantant des mécanismes de tolérance aux pannes. Nous avons présenté des systèmes à objets utilisant la notion de groupe pour mettre en place des mécanismes de redondance. Nous nous sommes inspiré de ces derniers pour réaliser un premier prototype implantant de la redondance active comme celle définie par Delta 4. Ce prototype utilise l'invocation sur groupe de COOL. Nous avons constaté que la version de COOL utilisée pour nos expérimentations n'était pas totalement adaptée **à nos besoins**. Le deuxième prototype associe à la redondance active des mécanismes de points de reprise. Faute de temps, ce dernier travail n'a pu être complètement mené à terme. Si nous avons montré qu'il était possible de restaurer l'état d'un automate Esterel et que l'on pouvait le resynchroniser avec les autres répliques de son groupe de vote, nous n'avons pas démontré que les coûts impliqués par ces mécanismes étaient compatibles avec les applications embarquées temps réel. Toutefois, nous avons pu constaté que l'horloge globale de Saturne simplifie la conception des protocoles et des algorithmes répartis. Si nous avons pu réaliser en si peu de temps ces deux prototypes, c'est essentiellement grâce à cette dernière propriété et à la facilité avec laquelle on peut développer des applications COOL.

L'étude des mécanismes de tolérance aux pannes sur Saturne est un sujet vaste. Nous pensons que deux aspects sont intéressants à étudier dans l'avenir : l'implantation de points de reprise basés sur l'enregistrement de l'état des automates Esterel et l'amélioration des mécanismes de détection de pannes. La difficulté soulevée par implantation de points de reprise est l'identification à la compilation des variables d'états d'un automate et l'adaptation du compilateur Esterel afin qu'il puisse fournir aux utilisateurs, un outil permettant de sauvegarder et restaurer l'état d'un automate. Enfin, les mécanismes de détection que nous avons utilisés restent sommaires, il serait intéressant d'en concevoir d'autres qui permettraient de détecter plus finement de quels types de pannes sont victimes les sites et les noyaux, par exemple savoir diagnostiquer l'occurrence de pannes transitoires.

Bibliographie

[1] F. Boniol M. Adelantado. Programming distributed reactive systems : a strong and weak synchronous coupling. CERT ONERA, Septembre 1993.

[2] L. Lin M. Ahamad. Checkpointing and rollback-recovery in distributed object based systems. pages 97–104. Proc. IEEE, Septembre 1990.

[3] D. Cummings L. Alkalaj. Checkpoint/rollback in a distributed system using coarse-grained dataflow. Jet Propulsion Laboratory, California Institute of Technology, Proc. IEEE, 1994.

[4] G. Berry. The constructive semantics of pure esterel. Ecole des Mines de Paris, Sophia-Antipolis, 1995.

[5] G. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[6] G. Berry, P. Couronné, and G. Gonthier. Programmation synchrone des systèmes réactifs: le langage Esterel. *Techniques et Sciences de l'Informatique*, 6(4), 1987.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[8] M. Adelantado F. Boniol. Programming communicating distributed reactive automata : the weak synchronous paradigm. CERT ONERA, International Conference on Decentralized and Distributed Systems, Palme de Mallorca, Spain, Septembre 1993.

[9] M. Adelantado F. Boniol. SATURNE : un modèle de description de systèmes multi-agents temps réel et intelligents. CERT ONERA, Mai 1994.

[10] A. Bouali. XEVE : An esterel verification environment. CMA-Ecole des Mines de Paris, Rapport Interne, Juin 1996.

[11] F. Boulanger. Intégration de modules synchrones dans la programmation par objets. Thèse de doctorat, PARIS XI Orsay, Décembre 1993.

[12] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Fujitsu Laboratories Ltd, ACM Computing Surveys, Mai 1991.

[13] E. Audureau P. Enjalbert L. Farinas Del Cerro. Logique temporelle : sémantique et validation de programmes parallèles. Edition Masson, 1990.

[14] CISI INGENIERIE. Esterel V3 : language reference manual. Agence Provence ouest, 1995.

[15] Kenneth Birmann Timothy Clark. Performance of the isis distributed computing toolkit. Juin 1994. Technical report TR-94-1432.

[16] Kenneth Birmann Robert Cooper. The isis project : Real experience with a fault tolerant programming system. ACM Operating Systems Review, Avril 1991.

[17] M. Adelantado F. Boniol M. cubéro-castan B. Lécussan R. Porche V. David. Projet SATURNE : modèle de programmation et modèle d'exécution pour un système temps réel d'aide à la décision. CERT ONERA, 5th Euromicro Workshop on Real-Time, Oulu, Finland, Juin 1993.

[18] Partha Dasgupta, Richard J. Leblanc, Jr., and William F. Appelbe. The Clouds distributed operating system: Functional description, implementation details and related work. In *The 8th International Conference on Distributed Computer Systems*, pages 2–9, S. José CA (USA), June 1988. (IEEE).

[19] A. Bouali A. Ressouche V. Roy R. de Simone. The FCTOOLS user manual (version 1.0). INRIA, Rapport Technique numéro 191, Avril 1996.

[20] David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *Computer*, 21(12):57–69, Décembre 1988.

[21] Singhoff F. Mise en oeuvre du modèle Saturne sur CHORUS/COOL. Mémoire d'ingénieur CNAM, Centre de Paris, Septembre 1996.

[22] Y. Faure. SATURNE objet : une approche distribuée orientée objet des systèmes temps réel mixtes réactifs/interruptibles. ENSAE, Rapport de DEA, Juin 1996.

[23] J.B. Stefani G.S. Blair g. Coulson M. Papathomas P. Robin F. Horn L. Hazard. A programming model and system infrastructure for real-time synchronization in distributed multimedia systems. Centre National d'Etudes des Télécommunications (CNET), IEEE Journal on selected areas in communications, vol 14, no 1, Janvier 1996.

[24] B. Garbinato, X. Défago, R. Guerraoui, and K. R. Mazouni. Abstractions pour la programmation concurrente sur garf. La lettre du Transputer et des calculateurs parallèles, Juin 1994.

[25] B. Garbinato, R. Guerraoui, and K. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal*, pages 14–27, 1995.

[26] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed programming in GARF. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*. Springer Verlag, 1994. ftp://ftp-lse.epfl.ch/pub/private/publication/1993/OBDP.ps.

[27] P. Le Guernic T. Gautier. Dataflow to Von Neumann : the SIGNAL approach. INRIA-RENNES, Rapport numéro 1229, 1990.

[28] The Isis Group. Muts documentation. Février 1993.

[29] K. Shin D. Kandlur D. Kiskis P. Dodd H. Rosenberg A. Indiresan. A distributed real time operating system. IEEE Software, Septembre 1992.

[30] INRIA. Rapport d'activité scientifique 1995 du projet MEIJE. Parallélisme, synchronisation et temps réel, 1995.

[31] Kenneth Birmann Thomas A. Joseph. Reliable communication in the presence of failure. ACM Trans. on Computer Systems, Février 1987.

[32] K. Chandy L. Lamport. Distributed snapshots : determining global states of distributed systems. volume 3, pages 63–75. ACM Trans. on Computer Systems, 1985.

[33] P. Weiss X. Leroy. Le langage Caml. Interédition, 1993.

[34] INMOS Limited. OCCAM 2 reference manual. Prenticel Hall, 1987.

[35] Sean Landis Silvano Maffeis. Building reliable distributed systems with CORBA. Cornell University, Isis distributed Systems Inc, 1994.

[36] Silvano Maffeis. A flexible system design to support object groups and object oriented distributed programming. Departement of Computer Science, University of Zurich, 1993.

[37] Silvano Maffeis. Adding group communication and fault tolerance to CORBA. Departement of Computer Science, University of Zurich, Juin 1995.

[38] Silvano Maffeis. Run time support for object oriented distributed programming. Université de Zurich, Février 1995.

[39] P. Le Guernic T. Gautier M. Le Borgne C. Le Maire. Programming real time applications with SIGNAL. INRIA-RENNES, Rapport numéro 1446, 1991.

[40] Y. AMir D. Dolev S. Kramer D. Malki. Transis : A communication sub-system for high availability. Proc. IEEE, 1992.

[41] C. André H. Boufaied D. Gaffé J.P. Marmorat. Environnement pour la programmation synchrone des systèmes réactifs. pages 27–41. Présenté à Real Time & Embedded Systems, Paris , Laboratoire Informatique, université de Nice Sophia Antipolis, CMA-Ecole des Mines de Paris, Janvier 1996.

[42] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Programmation d'une application distribuée résistante aux pannes avec l'environnement GARF. In *Proceedings of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence*. Maghrebian Information Processing Society, 1994.

[43] L. Jategaonkar Jagadeesan C. Puchol J.E. Von Olnhausen. Safety property verification of esterel programs and applications to telecommunications software. Proc. of the Seventh Conference on Computer Aided Verification, Juillet 1995.

[44] OMG TC Document 93.7.2. Object request broker architecture. 1993.

[45] G. W. George E. Kryal OSAF FORUM Draft. The perception and use of standards and components in embedded software development. Juillet 1996.

[46] N. Halbwachs P. Caspi P. Raymond D. Pilaud. Programmation et vérification des systèmes réactifs : le langage LUSTRE. Techniques et Sciences Informatiques, 1991.

[47] J.E. Pin. Théorie des automates finis. 1993.

[48] D. Harel A. Pnueli. On the developement of reactive systems. In Logic and Models of Concurrent Systems. Proc NATO Advanced Study Institute on Logics and Models for Verifications and Specification of Concurrent Systems, 1985.

[49] Z. Manna A. Pnueli. A hierarchy of temporal properties. Proc. of the 2th symph. ACM of principle of distributed computer, 1990.

[50] M. Adelantado F. Boniol R. Porche. Un modèle synchrone distribué et déterministe pour le temps-réel. CERT ONERA, Mai 1993.

[51] M. Adelantado F. Boniol R. Porche. Etude d'un modèle hiérarchique et structuré multi-synchrone pour la programmation de systèmes réactifs. CERT ONERA, Janvier 1996.

[52] M. Adelantado F. Boniol V. David B. Lecussan R. Porche. Predictability in distributed intelligent real-time systems. CERT ONERA, First IEEE Workshop on Parallel and Distributed Real-Time Systems, Newport Beach, California, Avril 1993.

[53] B. Garbinato R. Guerraoui and K. Mazouni. The garf library of dsm consistency models. In *Proc 6th ACM SIGOPS European Workshop*, 1994.

[54] T. M. Hickey D. Malki A. Vaysburd W. Vogels R. Renesse K. P. Birman B. Glade K. Guo M. Hayden. Horus: A flexible group communications system. Mars 1995.

[55] B. Folliot P. Sens P.G. Raverdy. Plate-forme de répartition de charge et de tolérance aux fautes pour applications parallèles en environnement réparti. pages 345–366. PARIS VI, Calculateurs Parallèles, Vol 7, 1995.

[56] M. Raynal. Synchronisation et état global dans les systèmes répartis. Chapitre 13, Editions Eyrolles, 1992.

[57] H. Kopetz H. Kantz G. Grunsteidl P. Puschner J. Reisinger. Tolerating transient faults in MARS. Proc. IEEE, Janvier 1990.

[58] D. Powell P. Martin D. Seaton. La tolérance aux fautes dans les systèmes répartis : l'approche Delta-4. Techniques et Sciences Informatiques, Février 1987.

[59] P.A. Barette A.MA Hilborne P.Verissimo L.Rodrigues P.G. Bond D.T. Seaton. The delta-4 extra performance architecture (XPA). 1990.

[60] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of Arjuna: A programming system for reliable distributed computing. Technical Report 298, University of Newcastle-upon-Tyne, Newcastle-upon-Tyne, England, Novembre 1989.

[61] Alexey Vaysburd Silvano Maffeis and Sophia Georgiakaki. A interoperable middleware infrastructure supporting object group communication. Departement of Computer Science, Cornell University, 1993.

[62] L. Leboucher J.B. Stefani. Admission control for end to end distributed bindings. Centre National d'Etudes des Télécommunications (CNET), Lecture Notes in computer science, Teleservice and Multimedia Communications, vol 1052, Novembre 1995.

[63] Kenneth Birmann Pat Stephenson. Fast causal multicast. Proc. of the ACM, Juin 1991.

[64] R. Renesse K. Birman R. Cooper B. Glade P. Stephenson. Reliable multicast between microkernels. 1992.

[65] Chorus Systèmes. Overview of the CHORUS distributed operating systems. Février 1991. CS/TR-90-25.1.

[66] Chorus Systèmes. CHORUS/COOL-ORB programmer's guide. Février 1996. CS/TR-96-2.1.

[67] A. Aho J. Ullman. Concepts fondamentaux de l'informatique. Edition DUNOD, 1993.

[68] Robbert van Renesse Kenneth P. Birman Robert Cooper. The horus system. Juillet 1993.

[69] D. Powell G. Bonn D. Seaton P. Verissimo. The Delta-4 approach to dependability in open distributed systems. Proc. IEEE, 1988.

[70] D. Powell M. Chérèque P. Reymier J.L. Richier J. Voiron. Active replication in Delta-4. Proc. IEEE, 1992.

[71] J.C. Fabre V. Nicomette T. Pérennou R. J. Stroud Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. pages 489–498, 1995.

[72] T. J. Mowbray R. Zahavi. The essential CORBA. OMG Document, 1995.