Tolérance aux fautes sur CORBA par protocole à métaobjets et langages réflexifs

Marc-Olivier Killijian

Année 2000 - N° d'ordre : 1641

THÈSE

présentée au
Laboratoire d'Analyse et d'Architecture des Systèmes
du Centre National de la Recherche Scientifique

en vue d'obtenir le titre de

Docteur de l'Institut National Polytechnique de Toulouse
Spécialité : Informatique et Télécommunications

par

Marc-Olivier Killijian

Ingénieur INSA

Tolérance aux Fautes sur CORBA par Protocole à Métaobjets et Langages Réflexifs

Soutenue le 19 janvier 2000 devant le jury :

Président	David POWELL
Rapporteurs	Charles CONSEL
	Rachid GUERRAOUI
Examinateurs	Shigeru CHIBA
	Jean-Charles FABRE
	Jean-Bernard STEFANI

Rapport LAAS N° 00022

LAAS - CNRS
7, avenue du Colonel Roche
31077 Toulouse Cedex 4

Thèse de Doctorat de Marc-Olivier Killijian

«Tolérance aux Fautes sur CORBA par Protocole à Métaobjets et Langages Réflexifs»

Résumé

L'objectif de cette thèse est la conception et l'implémentation d'un protocole à métaobjets adapté à la tolérance aux fautes d'objets CORBA. En effet, il n'existe pas, à ce jour, de protocole à métaobjets satisfaisant dans ce contexte. Le protocole que nous définissons permet, d'une part, le contrôle du comportement et de l'état interne des objets CORBA, et d'autre part, le contrôle des liens entre clients et serveur ainsi qu'entre objets et métaobjets, le tout de façon dynamique. L' implémentation proposée est adaptée à l'utilisation d'une plate-forme CORBA standard grâce à l'utilisation de langages ouverts et de réflexivité à la compilation : ces outils permettent de personnaliser le processus de compilation afin d'exhiber à l'exécution les informations nécessaires aux mécanismes de tolérance aux fautes. Un autre avantage de la réflexivité à la compilation est de permettre, de façon simple, d'assurer le respect de conventions de programmation grâce au filtrage du code source des applications. Ce protocole, bien intégré à CORBA, tire également profit, lorsque c'est possible, des éléments réflexifs fournis par le support d'exécution du langage. C'est le cas avec Java, par exemple, qui permet la sérialisation des objets, grâce à ses aspects réflexifs limités. Lorsque le support du langage n'est pas réflexif, comme pour C++ par exemple, la réflexivité à la compilation permet également de mettre en œuvre des techniques de sauvegarde et de restauration de l'état des objets. Les différentes propriétés de ce protocole à métaobjets sont illustrées par une proposition d'architecture CORBA permettant d'intégrer à l'application des mécanismes de tolérance aux fautes de manière très flexible. Les propriétés de cette approche sont une bonne séparation entre l'application et les mécanismes non-fonctionnels implémentés dans les métaobjets, l'aspect dynamique du lien entre objets et métaobjets, la composabilité et la réutilisation des mécanismes ainsi que la transparence pour l'utilisateur. Enfin, ce protocole à métaobjets est suffisamment générique pour tirer parti de l' ouverture, au sens de la réflexivité, des logiciels de base (système d'exploitation et middleware) de la plate-forme.

Mots-Clefs : Tolérance aux Fautes, Réflexivité, CORBA, Protocole à métaobjets, Architecture

«Fault-Tolerance on Corba using metaobject protocols and reflective languages»

Abstract

The goal of this dissertation is to design and implement a metaobject protocol adapted to fault-tolerance in Corba applications. No currently available metaobject protocol is satisfactory in this context. We define a protocol that enables dynamic control of (a) the behaviour and internal state of Corba objects, and (b) client/server and object/metaobject links. The implementation we propose is well adapted to a standard Corba platform thanks to the use of open languages and compile-time reflection. These tools allow the compilation process to be customized to obtain information that is necessary for fault-tolerance. Another benefit of compile-time reflection is it enables the enforcement of programming conventions thanks to the filtering of application source code. This protocol, well integrated with Corba, can also benefit from reflective properties of the underlying language runtime, such as the limited reflection provided by Java for object serialization. When the language runtime is not reflective, as for C++, compile-time reflection can be used to implement methods for saving and restoring the internal state of objects. The various properties of this metaobject protocol are illustrated in an architecture proposal which allows fault-tolerance mechanisms to be integrated with the application in a flexible way. This approach offers useful properties such as separation of concerns between the application and the non-functional mechanisms implemented as metaobjects, dynamic links between objects and metaobjects, composability and reuse of mechanisms, and user transparency. Finally, this metaobject protocol is sufficiently generic to take advantage of any openness, in a reflective sense, of the platform's underlying software (operating system and middleware).

Keywords: dependability, reflection, Corba, metaobject protocol, architecture

AVANT-PROPOS

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie Messieurs Alain Costes et Jean-Claude Laprie, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.

Je remercie également Messieurs Jean-Claude Laprie et David Powell, Directeurs de Recherche CNRS, responsables successifs du groupe de recherche Tolérance aux Fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

J'exprime ma profonde gratitude et toute mon amitié à Monsieur Jean-Charles Fabre, Chargé de Recherche CNRS, pour m'avoir parfaitement encadré durant cette thèse. Son rôle aura été primordial lors de ma formation ainsi que dans ce travail. Il aura su m'accompagner subtilement grâce à ses nombreuses qualités scientifiques et humaines.

Je remercie Monsieur David Powell pour l'honneur qu'il me fait en présidant mon jury de thèse, ainsi que :

- Monsieur Charles Consel, Professeur à l'Université de Rennes ;
- Monsieur Rachid Guerraoui, Professeur à l'Ecole Polytechnique de Lausanne (EPFL) en Suisse ;
- Monsieur Shigeru Chiba, Professeur à l'Université de Tsukuba au Japon ;
- Monsieur Jean-Charles Fabre, Chargé de Recherche CNRS au LAAS ;
- Monsieur Jean-Bernard Stéfani, Directeur de Recherche au Centre National d'Etudes des Télécommunications (CNET) ;

pour l'honneur qu'ils me font en participant à mon jury. Je remercie particulièrement Messieurs Charles Consel et Rachid Guerraoui qui ont accepté la charge d'être rapporteurs.

Au delà de son encadrement exemplaire, Jean-Charles Fabre (Après l'effort ...) a su créer autour de lui une équipe formidable avec laquelle j'ai pris un grand plaisir à travailler. Je tiens à les remercier chaleureusement : Juan-Carlos Ruiz-Garcia (¡Boneyour!), Laurent Blain (Méta-support), Eric Marsden (Australopithèque), Tanguy Pérénnou (Peluche Humaine) et Jean Fanchon (68's not dead).

Lors de nos nombreuses collaborations avec l'Université de Tsukuba, j'ai eu la chance de travailler le Professeur Shigeru Chiba (Mille-Feuilles) et son élève Michiaki Tatsubori (Monsta'Killa). Je tiens à les remercier pour leur aide et leur accueil chaleureux lors de mes deux séjours au Japon.

Je tiens également à remercier les membres du groupe TSF et du *LIS*, permanents, doctorants et stagiaires, ainsi que Joëlle Penavayre et Marie-José Fontagne pour leur constante disponibilité et leur gentillesse. Je réserve ici une mention particulière à Jean-Paul Blanquart (Space cow-boy) pour sa relecture complète et attentive de mon manuscrit, ainsi qu'à Agnan de Bonneval (Faut-qu'ch'te-parle) et à Vincent Nicomette (L'excellence du ... poulet) pour leurs conseils opportuns lors de la rédaction de cette thèse et de la préparation de la soutenance.

Nombreux sont ceux qui, à des titres divers, auront participé à l'aboutissement de ces travaux, qu'ils en soient ici remerciés : Christian Artigues, Salimeh Benhia, Hélène Dedryvère, Olfa Kaddour, Yannick Le Guédart, Philippe Torres et Eric Totel pour avoir accepté de relire ce mémoire ainsi que Jean Arlat, Yves Crouzet, Mohamed Kaâniche et Yvan Labiche pour m'avoir assisté lors des répétitions de ma présentation.

Mes remerciements s'adressent également à tous les membres des services *Informatique et Instrumentation, Documentation-Edition, Magasin, Entretien, Direction-Gestion, Réception-Standard* qui m'ont toujours permis de travailler dans d'excellentes conditions.

Enfin, bien sûr, je remercie chaleureusement tous ceux qui, en dehors du laboratoire, m'ont accompagné et soutenu : mes parents sans qui, évidemment, rien n'aurait été possible, toute ma raïa dont la liste est trop longue mais pour chacun de mes amis je réserve une pensée particulière, et Hélène pour son soutien sans faille, ses encouragements, son aide et sa bonne humeur.

SOMMAIRE

INTRODUCTION GÉNÉRALE

De nos jours, les systèmes informatiques sont par nature distribués, l'usage d'Internet se démocratise, le commerce électronique est en plein essor, les bourses européennes et mondiales se regroupent en réseaux, etc. Les applications développées sur ces systèmes distribués, si elles ne sont pas critiques en termes de vies humaines, le sont souvent d'un point de vue économique. La disponibilité de ces applications est très importante car leurs usagers sont répartis sur toute la surface du globe, et que, par conséquent, les services qu'elles assurent doivent souvent être rendus 24 heures sur 24.

Malheureusement, assurer cette disponibilité n'est pas aisé. En effet, les systèmes utilisés sont très hétérogènes en termes de matériel et de logiciel, les applications sont de plus en plus complexes, et il est parfois nécessaire d'intégrer d'anciennes applications (*legacy*) aux nouvelles, etc. Récemment, des outils, notamment issus du monde orienté-objet, ont fait leur apparition pour tenter de résoudre ces différents problèmes d'hétérogénéité et de complexité des systèmes. Ces outils, par exemple CORBA ou Java, permettent de faire abstraction de la plate-forme utilisée pour le développement de ces systèmes. Néanmoins, la disponibilité de tels systèmes ne peut être assurée qu'en utilisant des techniques de sûreté de fonctionnement, et plus particulièrement de tolérance aux fautes.

Les notions de tolérance aux fautes et de système distribué sont, en effet, deux concepts complémentaires. Comme décrit dans [Powell 1998], les systèmes distribués nécessitent des mécanismes de tolérance aux fautes et la tolérance aux fautes tire profit des architectures distribuées, pour pouvoir répliquer les applications, par exemple. Ce lien fort est à l'origine de nombreux systèmes se proposant d'implémenter des mécanismes de tolérance aux fautes dans une architecture distribuée. Les différences fondamentales entre ces divers systèmes distribués tolérant aux fautes sont, d'une part, la couche du système dans laquelle sont intégrés les mécanismes de tolérance aux fautes et, d'autre part, la technique utilisée pour faire le lien entre l'application et les mécanismes.

Plus récemment, le concept de réflexivité, et plus particulièrement de protocole à métaobjets, a été utilisé pour séparer mécanismes et application de façon claire et indépendante. Le système FRIENDS [Fabre et Pérennou 1998] utilise un protocole à métaobjets fourni par un compilateur spécifique, OpenC++

[Chiba et Masuda 1993], afin de permettre le contrôle des objets de l'application au sein de métaobjets. Ces métaobjets implémentent différentes mécanismes de tolérance aux fautes. Ces travaux ont fait apparaître certaines limites, essentiellement dues au protocole à métaobjets utilisé : difficulté à contrôler l'état des objets complexes, aspect statique du lien objet-métaobjet, etc.

Les travaux originaux présentés dans cette thèse, proposent d'utiliser l'ouverture des langages à objet, et en particulier la réflexivité à la compilation (une métaclasse contrôle le processus de compilation des classes de l'application) afin de concevoir et de mettre en oeuvre un protocole à métaobjets dédié à la tolérance aux fautes des objets CORBA. En effet, à partir des leçons tirées des travaux précédents suivant une approche réflexive, nous proposons de concevoir un protocole à métaobjets adapté à la tolérance aux fautes qui, de plus, prenne en compte les spécificités de l'architecture et du modèle à objet spécifiés par CORBA. Ce protocole à métaobjets permet de contrôler le comportement et l'état des objets CORBA, de prendre en compte les relations spécifiques entre clients et serveurs CORBA, et enfin, d'utiliser la notion d'interface pour rendre dynamique le lien entre objets et métaobjets.

Le chapitre I présente les notions nécessaires à la compréhension de cette thèse, à savoir le modèle objet, l'architecture Corba, ainsi que les concepts de réflexivité et de protocoles à métaobjets. Le chapitre II propose une évaluation des différentes approches disponibles actuellement pour la tolérance aux fautes dans les systèmes distribués à base d'objets. Le chapitre III propose d'étudier, dans un premier temps, quelques mécanismes de tolérance aux fautes afin de déterminer les informations que leur mise en oeuvre nécessite. Nous y étudions ensuite les différents moyens d'obtention de ces informations en fonction du degré de réflexivité de la plate-forme utilisée. Enfin, nous définissons un protocole à métaobjet qui prend en compte ces différentes informations et nous proposons une implémentation de ce protocole à métaobjets dans le chapitre IV. Cette implémentation, grâce à l'utilisation de langages réflexifs, est possible sur une plate-forme «boîte-noire» (système, ORB et langage). Enfin, les propriétés du protocole à métaobjets ainsi implémenté sont illustrées au sein d'une architecture adaptable où quelques mécanismes de tolérance aux fautes simples sont réalisés. La description de cette architecture ainsi que la présentation de quelques relevés de performances font l'objet du chapitre V.

CHAPITRE I
SYSTÈMES À OBJETS, RÉFLEXIVITÉ ET PROTOCOLES À MÉTAOBJETS

Ce chapitre a pour vocation essentielle de présenter les concepts de base que nous allons ensuite utiliser dans le développement de cette thèse, à savoir les systèmes distribués à objets et les protocoles à métaobjets. En effet, suite à l'émergence rapide des technologies objets, celles-ci se sont révélées particulièrement adaptées à la conception et au développement des systèmes distribués. Leur évolution, depuis leur origine dans les années 1960 jusqu'à nos jours, a mené à la création de plusieurs technologies et à la définition de certaines normes, telle CORBA, que nous présentons respectivement dans les sections I.1 et I.2 de ce chapitre. D'autre part, les protocoles à métaobjets, issus des théories sur la réflexivité appliquées aux systèmes orientés-objet, ont prouvé qu'ils permettaient de développer séparément applications et mécanismes non fonctionnels, notamment des mécanismes de sûreté de fonctionnement. Nous présentons ces notions en section I.3.

I.1 Modèle à objets

Avant d'étudier les systèmes distribués à objets à proprement parler, nous allons définir précisément ce que nous entendons par objet. Dans cette section nous présentons aussi les propriétés attendues d'un modèle de programmation par objet. Nous insistons sur celles qui nous sont utiles dans un contexte de tolérance aux fautes, et sur les caractéristiques potentiellement dangereuses qu'elles présentent dans un tel contexte.

I.1.1 Concepts de base

Historiquement les technologies à objets sont apparues pendant les années 1960 dans deux domaines de recherche distincts : pour la simulation de systèmes réels, ainsi que pour la modélisation du raisonnement en intelligence artificielle. A partir de ces recherches parallèles, on peut distinguer deux familles de langages orientés-objet : les langages de classes et les langages acteurs.

Dans le domaine de l'intelligence artificielle, l'objectif était d'exprimer un raisonnement de manière à l'implanter dans des robots. Pour ce faire, on a imaginé un modèle où plusieurs petites entités coopèrent entre elles en communiquant ; le concept d'acteur était né. Un acteur est une entité qui encapsule des données et les opérations qui s'y rapportent. Dans le modèle de Gul Agha [Agha 1990], chaque acteur est actif et peut interagir avec d'autres acteurs à travers un port où il envoie et reçoit des messages. Les **langages acteurs** sont à la base d'une forme particulière d'interaction entre objets sur laquelle nous reviendrons : la délégation [Lieberman 1986].

En ce qui concerne la simulation, les systèmes réels modélisés étant de plus en plus complexes, le langage Simula fut le premier à proposer de décomposer un problème en sous-problèmes indépendants grâce à l'introduction du concept de classe. Plus tard, et dans d'autres domaines, de nombreux langages dits **langages de classes** ont suivi cette voie : Smalltalk, Eiffel et Java, par exemple. On notera que le C++, auquel nous nous intéressons particulièrement dans ce manuscrit, est un hybride entre langage de classes et langage structuré. Il a, en effet, hérité des particularités du langage C, tout en proposant un modèle à objets issu des langages de classes. Parmi ces deux familles de langages orientés-objet, c'est surtout cette dernière catégorie (les langages de classes) qui a émergée et à laquelle nous nous intéressons ici.

Les technologies à objets sont maintenant partie intégrante du Génie Logiciel. Elles permettent d'apporter au développement logiciel des propriétés [Meyer 1997] favorisant la réutilisation, la portabilité, l'extensibilité, l'interopérabilité, etc. De nombreuses méthodes de spécification et de conception ont alors émergé : Shlaer et Mellor [Shlaer et Mellor 1991], Booch [Booch 1994], et récemment UML [Rumbaugh *et al.* 1998] qui est une tentative d'unification de ces différentes méthodes.

I.1.1.1 Types abstraits de données

L'histoire de l'informatique fourmille d'anecdotes où le format des données a posé ou posera problème. Un premier exemple pourrait être fourni par un système bancaire qui prévoit un nombre de caractères maximal pour le nom de ses clients. Un client qui posséderait un nom plus long que cette limite, verrait, sur son chéquier, son nom amputé de quelques lettres et pourrait avoir quelques problèmes pour justifier de son identité. Un second exemple caractéristique est le fameux bug de l'an 2000.

Dans les années 70, suivant la célèbre devise «diviser pour régner» qui pourrait se décliner ici en «séparer pour maîtriser», la notion de **types abstraits de données** est introduite dans les langages de programmation [Hoare 1972] [Liskov et Zilles 1974]. Cette théorie consiste à séparer d'une part, la définition des données et le code nécessaire à leur gestion, et d'autre part le code qui les utilise ; ceci afin de limiter l'impact qu'aurait une modification de l'implémentation physique d'un type de données sur une application ou un système tout entier. Des types abstraits de données sont définis, leur implémentation est cachée ; ils ne sont accessibles qu'à partir de fonctions bien définies, on dit que les données sont encapsulées.

L'**encapsulation** est une propriété très intéressante pour la sûreté de fonctionnement. En effet, elle permet de mieux contrôler l'accès aux données puisque celles-ci ne sont pas accessibles directement ; les fonctions d'accès aux données peuvent être enrichies de manière à éviter certaines fautes, par exemple en effectuant des contrôles sur les valeurs d'entrée.

I.1.1.2 Un modèle simple

Dans la suite de la thèse, nous utiliserons le modèle issu des langages de classes et en particulier celui du langage Eiffel [Meyer 1997]. Ce modèle est le suivant :

Une **classe** définit un ensemble de variables, appelées **attributs**, et un ensemble d'opérations, appelées **méthodes**. Chaque objet est une **instance** d'une classe. Les méthodes d'une classe peuvent être appliquées sur les instances de cette classe. Le principe d'encapsulation s'applique aux objets : les attributs d'un objet ne sont accessibles qu'à lui seul. Les méthodes représentent donc l'**interface** d'une classe et de ses instances.

Pour interagir avec un objet, seule la connaissance de son interface est nécessaire : en effet, l'**implémentation** des méthodes est cachée derrière cette interface. Interface et implémentation sont donc indépendantes du point de vue de l'utilisateur. Cette propriété permet d'assurer une bonne maintenabilité du système : l'implémentation d'une classe peut changer de façon transparente si son interface (et son comportement observable) est conservée.

I.1.1.3 Programmation par contrat

Dans certains langages comme Eiffel, grâce à l'encapsulation et à une sémantique des interfaces bien définies, le modèle peut être enrichi par des **assertions :** des conditions vérifiables sur les paramètres d'entrée ou de sortie des méthodes ainsi que sur l'état des objets ; selon les cas on parle d'invariants, de pré- ou de post-conditions. Cette approche a donné naissance à la notion de **programmation par contrat** ou encore **contrôles de vraisemblance** [Laprie *et al.* 1996] ; en effet, les assertions représentent les clauses d'un contrat passé entre l'objet et ses clients. Si l'une de ces clauses n'est pas remplie, par exemple un des paramètres d'une méthode est invalide, l'objet ne fournira pas son service. La violation du contrat, une fois détectée, doit être signalée et éventuellement traitée, on parle alors d'**exceptions** et de traitement d'exceptions.

La programmation par contrat est très intéressante pour la sûreté de fonctionnement, elle permet de disposer d'un mécanisme simple de détection et de confinement d'erreur. Il faut toutefois noter que ce mécanisme a de sérieuses limitations [Rabejac 1995].

I.1.2 Héritage, délégation et polymorphisme

Nous venons de voir qu'un modèle à objets simple pouvait nous fournir des propriétés intéressantes telles que : la portabilité, la maintenabilité et l'extensibilité grâce en grande partie au principe d'encapsulation. Nous allons étudier maintenant d'autres caractéristiques des technologies à objets qui ont pour but de permettre une meilleure réutilisation : l'héritage et la délégation.

I.1.2.1 Héritage

Améliorer la réutilisation consiste principalement à fournir une technique de programmation qui permette de réunir les fonctionnalités communes à plusieurs structures (classes). L'héritage est un outil essentiel de la réutilisation : l'idée consiste à définir une classe comme étant un raffinement d'une autre classe. Par exemple, supposons qu'une classe *Polygone* soit définie, nous pouvons alors définir une classe *Rectangle* qui hérite de *Polygone* ; *Rectangle* héritera alors des attributs et méthodes de la classe *Polygone*, par exemple de la méthode *surface*, ou du fait qu'un *Polygone* possède plusieurs côtés. On dira dans ce cas que *Rectangle* est une **sous-classe** ou une **classe fille** de *Polygone* alors que *Polygone* est la **super-classe** ou encore **classe mère** de *Rectangle*.

Bien qu'une sous-classe hérite des attributs et méthodes de sa super-classe, elle peut néanmoins redéfinir certains de ces éléments : par exemple, on sait que le calcul du périmètre d'un rectangle est plus simple que celui d'un polygone (bien que la méthode pour les polygones soit évidemment valide pour les rectangles) ; la classe *Rectangle* peut alors redéfinir la méthode *périmètre* et donc tirer parti de cette connaissance pour optimiser les calculs effectués.

En poussant plus loin le concept d'héritage on peut définir une classe *Carré* qui héritera de *Rectangle* et qui pourra encore raffiner les méthodes *périmètre* et *surface*. On aurait alors une hiérarchie d'héritage telle que celle de la figure 1. On peut alors dire que *Carré* est une sous-classe de *Polygone* et de *Rectangle*, mais elle n'est la classe fille que de *Rectangle*. Ainsi *Polygone* est une super-classe de *Rectangle* et de *Carré* mais elle n'est la classe mère que de *Rectangle* tout comme *Rectangle* l'est pour *Carré*.

Toujours dans le but d'améliorer la réutilisation, bien des langages de programmation orientés-objet permettent de définir ce que l'on appelle des **classes abstraites**. Une classe abstraite est une classe qui n'implémente pas la totalité de son interface ; c'est-à-dire que certaines de ses méthodes sont définies mais non implémentées. Dans ce cas, puisque ces classes ne sont pas totalement définies, on ne peut pas créer directement d'instances de celles-ci ; elle ne sont faites que pour être héritées. La réalisation d'une classe abstraite dans une classe fille permet d'aboutir à la notion de **classe concrète**, à partir de laquelle des instances peuvent être obtenues.

Il se trouve des cas où une classe hérite de plusieurs classes-mères, on appelle cela l'**héritage multiple**. Un exemple bien connu d'héritage multiple est celui des *Doctorants* ; en effet, ceux-ci sont souvent au sein de l'université à la fois *Enseignants*

et *Etudiants*, ils perçoivent un salaire et payent un droit d'inscription. Pour compliquer les choses, les *Enseignants*, tout aussi bien que les *Etudiants* sont des *Personnes* ; nous obtenons alors la hiérarchie d'héritage de la figure 1. On dit dans ce cas que la hiérarchie d'héritage est **à répétition** [Hill 1993].

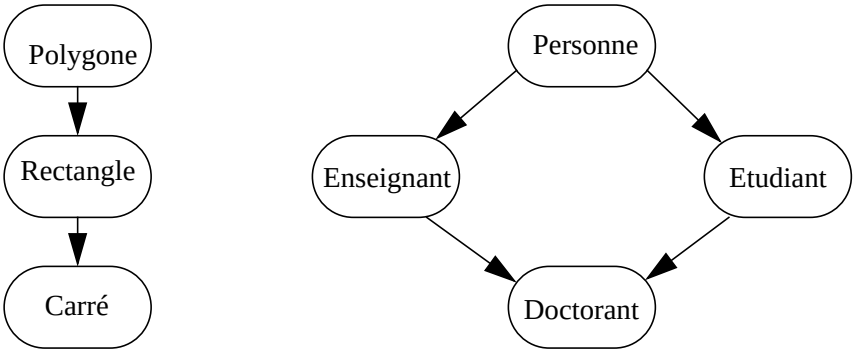

Figure 1 - Héritage simple et multiple

Bien qu'utile pour la réutilisation du code, l'héritage pose de nombreux problèmes théoriques et pratiques. Il peut arriver ainsi qu'une classe fille hérite de deux méthodes différentes portant le même nom. Plusieurs solutions sont apportées par les différents langages de programmation. Par exemple Eiffel [Meyer 1997] propose au programmeur de pouvoir renommer explicitement les méthodes qui posent problème, alors que C++ [Stroustrup 1992] propose de sélectionner implicitement une des méthodes en fonction de l'ordre de déclaration de la hiérarchie d'héritage.

Le cas particulier de l'héritage multiple à répétition pose lui aussi de nombreux problèmes : supposons dans la hiérarchie de la figure 1 que la classe *Personne* définisse un attribut *Adresse* qui corresponde à l'adresse personnelle de la *Personne*, et que la classe *Enseignant* redéfinisse cette *Adresse* comme étant l'adresse professionnelle utilisant le courrier interne de l'université. Un *Doctorant* recevra-t'il son courrier à son domicile comme une *Personne* (comme un *Etudiant* aussi) ou bien à son bureau comme un *Enseignant*? Une des solutions proposée est de nommer explicitement la classe à laquelle on désire s'adresser lors de l'utilisation d'une telle méthode, par exemple *Personne::Adresse* ou *Enseignant::Adresse*.

L'exemple précédent montre qu'il est parfois nécessaire d'avoir connaissance de la sémantique de certains attributs des super-classes, ce qui rend difficile la programmation par héritage. Bien que chacun de ces problèmes ait trouvé sa solution il en résulte tout de même des incohérences dans les modèles des différents langages de programmation. Lorsque les solutions proposées sont implicites, le programmeur peut facilement ne pas remarquer l'incohérence et le résultat obtenu peut être différent de celui escompté. Ceci peut conduire à l'introduction de fautes de conception dans le logiciel, c'est notamment la raison pour laquelle les bénéfices de l'héritage multiple restent encore aujourd'hui discutés dans le Génie Logiciel.

Pour pallier à ces problèmes, certains langages, comme Java [Gosling *et al.* 1996] par exemple, n'autorisent quant à eux que l'héritage simple de classe et l'**héritage multiple d'interfaces**. Dans ces langages on peut, en effet, déclarer séparément

interfaces et classes. Une interface définit des prototypes de méthodes et des constantes ; une interface peut donc être vue comme une classe totalement abstraite. L'héritage multiple d'attribut n'est donc pas possible, ainsi le problème souligné dans l'exemple de la figure 1 n'a pas lieu d'être.

Un autre argument en défaveur de l'héritage est que, bien souvent, les langages à objets assouplissent le principe d'encapsulation en cas d'héritage [Snyder 1986] : une classe fille peut accéder aux attributs de sa hiérarchie d'héritage. Nous verrons dans les chapitres suivants que l'encapsulation est, dans notre contexte, un concept très important, et que, par conséquent, de tels comportements sont indésirables.

I.1.2.2 Désignation des objets

Dans un programme orienté-objet, les objets ont besoin d'être représentés physiquement : de la mémoire leur est allouée et ils peuvent y stocker leurs attributs. Ces objets doivent aussi être manipulés en tant qu'entités abstraites, on doit pouvoir désigner un objet sans avoir à se soucier de son adresse mémoire. C'est le concept même de **référence :** une référence désigne un objet. Cette manière de référencer les objets est totalement compatible avec les types abstraits de données et le principe d'encapsulation.

Malheureusement, certains langages, dont C++, n'utilisent pas directement cette notion évoluée de référence mais celle moins abstraite de **pointeur** : un pointeur représente directement l'adresse mémoire d'un objet, ce qui permet un accès très efficace aux données. Cette façon de désigner les objets est, à juste titre, considérée comme extrêmement dangereuse pour l'encapsulation des données. En effet, un programmeur peut tout à fait accéder aux éléments internes de l'objet (attributs et méthodes) en utilisant une adresse mémoire, ce qui rend les contrôles d'accès impossibles à réaliser par le compilateur. D'autre part, en C++ l'**arithmétique de pointeur** est autorisée, accentuant ainsi ces problèmes potentiels.

I.1.2.3 Sous-typage, polymorphisme

La relation d'héritage entre deux classes peut-être vue comme une relation «est un» : un *Doctorant* est un *Etudiant* qui lui même est une *Personne*. En revenant aux types abstraits de données, on peut dire dans ces cas que la classe fille est un **sous-type** de sa classe mère, une classe correspondant à un type d'objets. Pour permettre plus de flexibilité et grâce au concept de référence, la plupart des langages à objets permettent alors le **polymorphisme**. Le polymorphisme est une fonctionnalité qui permet à une entité de prendre plusieurs formes, en l'occurrence une référence qui peut être attachée à plusieurs types d'objets, plusieurs classes ; on parle alors de **liaison dynamique** ou d'édition de lien retardée [Hill 1993]. Par exemple, une référence sur un objet *Personne* peut tout aussi bien représenter un *Enseignant* qu'un *Doctorant* ou encore un *Etudiant* ; un tableau de *Personnes* peut contenir à la fois des objets de ces trois classes, puisque l'un quelconque de ces objets «est une» *Personne*.

I.1.2.4 Composition, délégation

On appelle **composition** d'objets le fait qu'un des attributs d'un objet soit une instance d'objet ou une référence sur un autre objet. Par exemple (cf. figure 2) une classe *Automobile* pourra contenir des attributs de type *Roues*, *Moteur*, *Carrosserie*, etc. Ainsi, la classe **contenant** pourra se servir de l'instance **contenue** pour réaliser son propre comportement : l'*Automobile* utilisera les méthodes de *Moteur* pour accélérer ou décélérer, celles de *Roues* pour avancer ou reculer et les attributs de *Carrosserie* pour définir sa couleur.

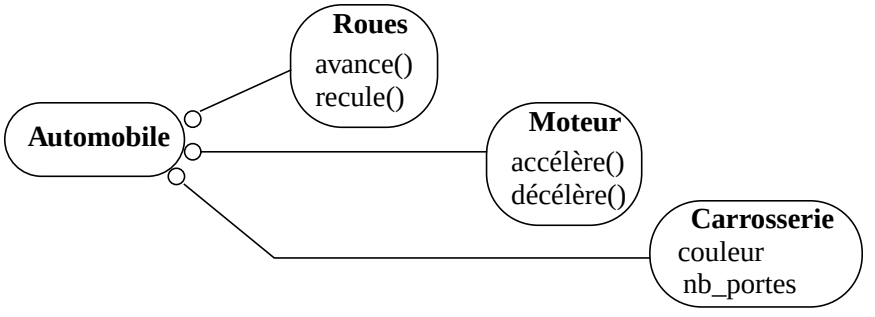

Figure 2 - Exemple de composition

Issue des langages acteurs, la délégation permet à un acteur qui ne sait répondre à un message, de déléguer le traitement de ce message à un autre acteur appelé son mandataire (*proxy*). Pour la réutilisation, la **délégation** est un moyen de rendre la composition aussi puissante que l'héritage [Lieberman 1986], [Stein 1987]. La délégation est toutefois plus souple que l'héritage : un objet peut choisir dynamiquement un mandataire pour traiter un message, alors que, pour la plupart des langages, la hiérarchie d'héritage est définie statiquement.

On utilise aujourd'hui intensivement la délégation dans les langages de classes. En effet, celle-ci permet de modéliser des interactions complexes entre les objets, elle est devenue incontournable lors de la résolution de certains problèmes récurrents comme c'est le cas pour les pièces de conception (design patterns) [Gamma *et al.* 1995] : fabrique d'objets abstraite, interpréteur, par exemple. Cependant, la délégation partage le même inconvénient que le polymorphisme : la complexité due ici au caractère dynamique du système rend les programmes plus difficiles à comprendre, et en conséquence à maintenir.

I.2 CORBA

De nos jours, bien des applications et systèmes développés sont des systèmes distribués : applications bancaires, commerce électronique, travail de groupe, Internet, etc. Il n'est quasiment plus d'application qui ne soit distribuée. Les réseaux et la communication ont pris une telle ampleur, que plus personne aujourd'hui, ne

peut se permettre de développer une application isolée, qui ne soit pas ouverte, qui n'utilise pas des services distribués. Même les jeux vidéos sont maintenant toujours prévus pour fonctionner en réseau, avec de multiples joueurs.

Nous allons étudier dans cette section pourquoi et comment les systèmes distribués sont dorénavant développés par des technologies à objets. Nous présenterons aussi les technologies les plus répandues dans le domaine des systèmes distribués à objets : CORBA et ses concurrents.

I.2.1 Introduction : des objets aux systèmes distribués à objets

L'utilisation des technologies à objets pour le développement des systèmes distribués a créé une révolution dans la façon de concevoir, implémenter et maintenir les applications. En effet, le concept d'objet s'applique très bien aux systèmes distribués : un objet fournit un service à d'autres objets tout comme un serveur fournit un service à ses clients à travers son interface.

Dans les langages à objets standards, tels que Smalltalk ou encore C++, les objets ne sont, en général, visibles qu'à l'intérieur d'un programme. A l'inverse, les objets distribués sont accessibles à travers un réseau par des clients distants, et ce, de façon transparente. Un protocole d'invocation de méthodes est donc nécessaire pour que les objets puissent interagir à distance. Grâce à un tel protocole on peut concevoir une application répartie comme un ensemble d'objets qui coopèrent à travers un réseau.

Les systèmes distribués à base d'objets, les abstractions qu'ils permettent et les outils qui les supportent, présentent de nombreux avantages [Orfali *et al.* 1996] :

- *Interopérabilité des langages* : grâce à la notion d'interface, un client et un serveur peuvent coopérer quel que soit le langage dans lequel ils sont implémentés.
- *Interopérabilité des systèmes* : au delà de la transparence du langage, la plateforme matérielle aussi bien que le réseau physique ou les protocoles de communication sont rendus transparents au programmeur.
- *Maintenabilité et extensibilité* : clients et serveurs sont développés séparément et de manière totalement indépendante ; ils peuvent donc évoluer parallèlement, ainsi la maintenabilité et l'extensibilité du système est accrue.
- *Coexistence* : on est capable d'encapsuler d'anciennes applications développées dans des langages non orientés-objet, tels que Cobol par exemple, de manière à les présenter sous forme d'objets distribués et, ainsi, les rendre plus facilement accessibles.
- *Mise à l'échelle* : La décomposition en objets de granularité fine permet de fournir des applications plus adaptées aux besoins de l'utilisateur plutôt que des applications dont l'ensemble des nombreuses fonctions n'est pas nécessaire à chacun d'entre eux (comme les traitements de textes ou les applications de gestion). Un éditeur de logiciel peut alors proposer à ses clients un catalogue de fonctionnalités. Chacune de ces fonctions est disponible dans un composant (un objet), et celui-ci est intégré à la demande au système global.

Aujourd'hui, les systèmes distribués sont devenus très complexes, et très hétérogènes. Les applications qui sont mises en œuvre sur de tels systèmes sont d'une ampleur difficilement imaginable il y a encore quelques années. De plus ces applications sont, bien souvent, très critiques en termes économiques ou encore en termes de vies humaines. Afin de pouvoir raisonner correctement à propos de tels systèmes et de telles applications, un modèle abstrait est nécessaire.

Un système distribué est composé d'un ensemble de machines (ou nœuds), connectées entre elles par un canal de communication. Ces machines exécutent des processus qui utilisent ce canal de communication pour collaborer dans le but de remplir une certaine tâche. Les caractéristiques du système ainsi constitué dépendent donc de celles de ses différents constituants, nœuds et canal de communication. Nous reviendrons sur ces points au chapitre II.

Un système distribué à objets (cf. figure 3) est basé sur la définition précédente hormis que les processus sont des objets, c'est-à-dire qu'ils ont une interface à travers laquelle ils coopèrent, de façon transparente, par invocation de méthode.

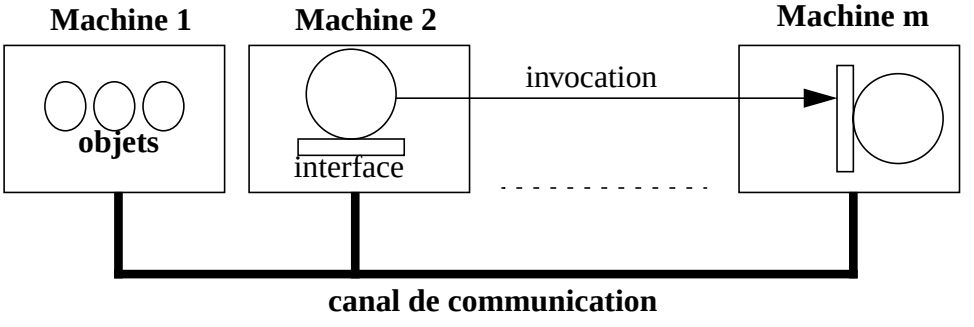

Figure 3 - Modèle de système distribué à objets

I.2.2 Architecture de CORBA

Suite au développement exponentiel des systèmes distribués, de nombreuses technologies et protocoles propriétaires ont vu le jour ; cette hétérogénéité dans les systèmes posait de nombreux problèmes d'intégration entre les différents systèmes. Le monde des systèmes distribués avait besoin de standards permettant à ceux-ci d'interopérer. L'Object Management Group (OMG), un consortium d'industriels impliqués dans les technologies orientées-objet, a donc été crée en 1989 pour développer les spécifications d'une architecture distribuée standardisée, l'Object Management Architecture (OMA) [OMG 1998a]. La caractéristique clé qui était recherchée était l'indépendance vis-à-vis des plate-formes, réseaux et langages pour permettre une portabilité accrue des applications développées ainsi qu'une interopérabilité entre les différents systèmes.

CORBA (Common Object Request Broker Architecture) est le composant central de cette architecture. On peut le considérer comme le réseau (ou le bus) qui permet aux différentes entités de communiquer. Dans la pratique, on utilise souvent le terme

CORBA pour désigner aussi bien l'architecture que le bus à objets, ou encore le modèle dans son ensemble. Dans cette section nous nous attachons à présenter le modèle architectural de l'OMA ainsi que les caractéristiques du modèle que nous utiliserons ultérieurement.

L'architecture de l'OMA est composée de quatre entités principales (cf. figure 4) :

- *Le bus à objets (Object Request Broker - ORB)* est le composant central de l'architecture. Il permet aux différentes entités de communiquer entre elles en assurant la distribution des requêtes entre émetteurs et récepteurs. Certaines entités lui sont associées afin de fournir certaines fonctionnalités : un dépositaire d'interfaces, un dépositaire d'implémentations, etc.
- *Les services (Object Services)* fournissent des fonctions indispensables telles que le nommage des objets, un gestionnaire d'évènements, etc.
- *Les outils communs (Common Facilities)* définissent des outils transversaux aux applications tels que la gestion des tâches ou du système.
- *Les objets applicatifs (Application Objects)* sont les objets de l'application distribuée. Ils sont régis par un modèle à objets simple qui définit un objet comme une entité encapsulée, à l'identité immuable et distincte. On ne peut accéder à cet objet qu'à travers son interface qui est prédéfinie à la conception.

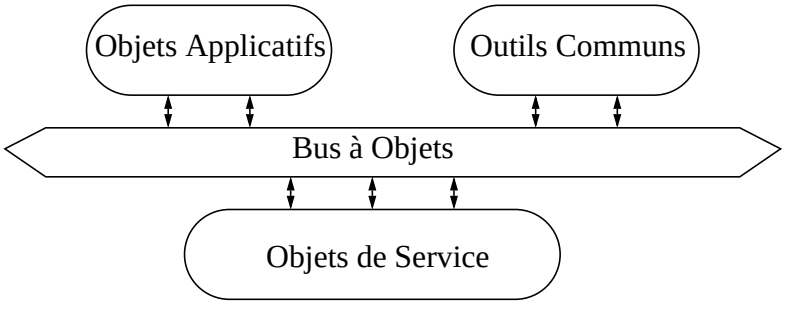

Figure 4 - L'architecture de CORBA (OMA)

I.2.2.1 Le bus à objets

Le bus à objets est l'abstraction la plus importante de l'architecture CORBA : il permet à un objet client d'invoquer une méthode sur un objet serveur où qu'il soit localisé. En effet, dans CORBA, on accède aux objets à travers des références. Celles-ci sont typées et permettent ainsi à l'ORB d'effectuer à leur propos certains contrôles de types. De cette manière, pour utiliser les services d'un objet, il suffit d'obtenir une référence sur celui-ci et de connaître ou de découvrir dynamiquement son interface.

I.2.2.2 Communication entre client et serveur

L'interface d'un objet est décrite dans un langage particulier de description d'interfaces, l'Interface Definition Language (**IDL**). L'utilisation d'un tel langage permet de décrire des interfaces indépendamment de la manière et du langage utilisés pour

l'implémentation des objets. Un client accédera donc au serveur à travers son interface ; les requêtes qu'il enverra au serveur correspondront aux méthodes décrites dans cette interface. Une entité, le **stub** (cf. figure 5), est chargée de traduire une requête issue du client en requête CORBA. Côté serveur, une autre entité, le **skeleton**, est chargée de traduire les requêtes CORBA dans le langage d'implémentation du serveur.

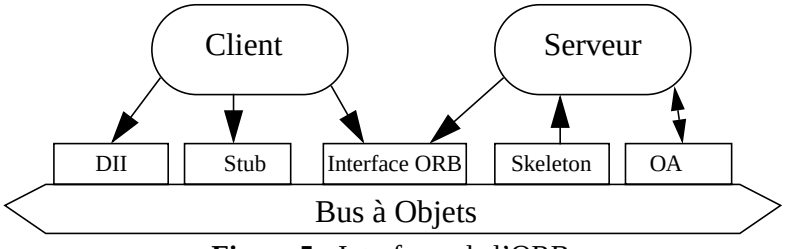

Figure 5 - Interfaces de l'ORB

Ces deux entités, stub et skeleton sont générées automatiquement par un compilateur d'IDL et apportent un niveau intermédiaire qui permet de rendre les communications indépendantes du langage. Il existe des compilateurs IDL vers de nombreux langages tels que C, C++, Java ou encore Ada. Ainsi le développeur n'a pas à se soucier du langage utilisé pour l'implémentation du serveur (respectivement du client) ; CORBA sert de médiateur.

Pour être utilisés, stubs et skeletons doivent être connus statiquement lors du développement du système. Cependant, CORBA fournit parallèlement un moyen de découvrir et d'utiliser des serveurs qui n'étaient pas connus lors du développement. L'*interface d'invocation dynamique (**DII**)* permet de découvrir et d'utiliser, au moment de l'exécution, l'interface des objets. Grâce à la DII, le système est hautement extensible, car un client peut utiliser un service qui n'existait pas lors de son développement. La DII utilise intensivement les services du *dépositaire d'interfaces (Interface Repository)* qui, pour sa part, stocke les interfaces de tous les objets du système.

I.2.2.3 Services

Les services définis par l'OMG implémentent des fonctionnalités essentielles dans de nombreuses applications distribuées. Ces services sont nombreux, nous présentons uniquement ici ceux qui nous intéresseront dans cette thèse. On notera toutefois que la majeure partie des services présentés n'est pas implémentée dans les produits actuellement sur le marché des ORBs ; certains restent aujourd'hui à l'état de simple spécification.

- Nommage : la fonction de nommage permet d'obtenir une référence sur un objet à partir de son nom ; ce service [OMG 1998a] participe grandement à la flexibilité du système. Un client qui désire utiliser un certain serveur comme par exemple «Serveur de fichiers» peut obtenir une référence sur celui-ci où qu'il soit localisé sur le réseau. Le serveur en question peut aussi se déplacer de machine en machine et rester accessible à tout moment par ses clients à travers son nom.
- *Objet Persistant* : la persistance est la propriété d'un objet qui conserve son état. D'une exécution sur l'autre, cet objet conservera son état ; c'est une propriété très importante pour les bases de données par exemple. Un tel service [OMG 1998d] est très proche du concept de sauvegarde sur support stable pour la tolérance aux fautes : l'état de l'objet doit être sauvegardé, afin qu'à la prochaine exécution du serveur, celui-ci puisse reprendre son exécution à partir de l'état dans lequel il a été arrêté.
- *Externalisation* : très proche du service précédent, le service d'Externalisation [OMG 1998b] définit un protocole qui permet de transmettre l'état d'objets à travers l'ORB. Les contextes d'utilisation sont nombreux et vont de la mobilité des objets à la tolérance aux fautes. Il faut toutefois noter qu'aucun de ces deux services, objet persistant ou externalisation, ne définit la façon dont l'état des objets est obtenu. Ils définissent simplement des protocoles qui régissent les interactions entre ces services et les objets qui désirent les utiliser. Comme nous le verrons dans le prochain chapitre, obtenir l'état des objets est une fonction nécessaire et non triviale, que ce soit pour la tolérance aux fautes, la migration des objets, ou encore leur persistance.
- *Intercepteurs* : en cours de spécification [OMG 1998e], ce service a un très grand intérêt dans le cadre de cette thèse. En effet, il définit la façon dont une entité peut contrôler les interactions entre un objet CORBA et son environnement. Autrement dit, cette entité peut contrôler, par exemple, l'obtention de références ou l'invocation de méthodes sur un objet particulier. Comme nous le verrons dans le prochain chapitre, ces informations sont de la plus haute importance en ce qui concerne la réplication des objets.
- *Tolérance aux Fautes* : consciente des besoins en tolérance aux fautes des systèmes répartis, l'OMG essaye de spécifier un service de réplication d'objets [OMG 1998c]. A ce jour, l'organisation collecte les propositions de spécification de ses partenaires. L'adoption de la spécification n'est pas encore à l'ordre du jour.
- *Meta-Object Factory* : contrairement à ce que son nom peut laisser entendre, la Meta-Object Factory définie par l'OMG [OMG 1997] n'est pas liée à la notion de métaobjet au sens ou nous l'entendons. Elle est plutôt un dépositaire d'informations à la manière d'un dépositaire d'interfaces. Les informations qui y sont stockées sont relatives à la structure des objets : attributs, méthodes, types des

arguments, liens de composition ou d'héritage, etc. Ces informations sont donc réflexives, on peut les qualifier de méta-informations structurelles. Nous reviendrons plus en détail sur ces notions de réflexivité et de méta-information au paragraphe I.3 et au chapitre II.

I.2.3 Alternatives à CORBA

Bien que CORBA s'impose en tant que standard dans l'élaborations de systèmes distribués orientés-objet, certaines alternatives existent avec leurs avantages et leurs inconvénients. On notera que l'inconvénient principal de ces technologies est de ne pas être toujours compatible avec CORBA lui-même.

I.2.3.1 Java et les technologies associées

Java s'est récemment imposé comme un des meilleurs langages de programmation orienté-objet. En effet, il dispose de nombreux avantages comme par exemple :

- une portabilité à priori sans limite puisqu'il est basé sur l'utilisation d'une machine virtuelle et d'un interpréteur ;
- il est très adapté à la programmation distribuée, notamment au travers de l'Internet ;
- son modèle à objet est très propre, contrairement à celui de C++ qui hérite de certaines caractéristiques du langage C.

Aux avantages issus du langage de programmation proprement dit, s'ajoutent ceux issus de deux technologies développées autour de celui-ci, JavaRMI et les JavaBeans :

- JavaRMI (pour Java *Remote Method Invocation*) est un moyen de faire interopérer, de façon transparente, différents objets Java au sein d'un réseau. On peut faire le parallèle avec le bus à objet de l'architecture CORBA : transparence de la localisation grâce à l'utilisation de références, nommage des objets, etc. De plus, dans une récente évolution, JavaRMI, grâce à l'utilisation du protocole de transport IIOP, peut interopérer avec les systèmes CORBA.
- JavaBeans quant à lui est un système qui permet à des composants logiciels d'être intégrés facilement : des outils sont fournis pour les composer telles des briques de Lego. On peut les interroger pour découvrir dynamiquement leur interface. En ce sens, ils sont équivalents à ce que permet l'interface d'invocation dynamique de CORBA.

Ces technologies représentent une bonne alternative à CORBA pour qui se contente du langage Java. Il est de plus important de noter que leur utilisation s'avère être bien plus simple que celle d'un ORB.

I.2.3.2 DCOM et OLE

DCOM et Network OLE sont deux technologies développées par Microsoft pour la conception et l'intégration de composants logiciels :

- DCOM [Rogerson 1997] définit un modèle d'objets distribués et remplit les fonctions d'ORB telles que définition d'interfaces, gestion de références, invocation dynamique, etc. Ce système est intégré aux systèmes d'exploitation de Microsoft et leur est propriétaire.
- Network OLE permet d'intégrer des composants entre eux à la manière des JavaBeans pour faciliter le développement et la réutilisation de code. En effet, dans les systèmes d'exploitation et de fenêtrage graphiques comme Windows, de nombreux éléments sont communs aux applications : le presse-papier, l'impression de documents et aujourd'hui, les services multimedias ou liés à l'Internet.

Ces deux technologies peuvent donc être comparées à CORBA mis à part qu'elles ne sont disponibles que sur les systèmes d'exploitation Windows. Contrairement aux technologies Java, elles permettent l'interopérabilité des langages et non l'interopérabilité des systèmes.

I.3 Réflexivité et protocoles à métaobjets

De nombreuses classes d'applications nécessitent des propriétés totalement indépendantes de leurs fonctionnalités, comme la persistance ou la sécurité. On qualifie parfois ces propriétés de non fonctionnelles car elles sont orthogonales à l'application. Des outils issus du modèle orienté-objet ont été développés pour traiter certains mécanismes orthogonaux récurrents comme les «fabriques abstraites d'objets» ou encore les «adaptateurs». Ces outils sont des modèles de conception qui utilisent la délégation et l'héritage afin d'augmenter leur réutilisabilité et leur généralité ; on appelle ces outils les «Design Patterns» [Gamma *et al.* 1995].

Une autre solution pratique pour l'implémentation de mécanismes orthogonaux à l'application est l'utilisation de la réflexivité [Stroud 1993], particulièrement sous la forme de protocole à métaobjets. En effet, ces notions ont prouvé qu'elles permettent non seulement d'implémenter ces mécanismes d'une façon élégante et efficace mais surtout qu'elles fournissent une grande flexibilité, tout en étant transparentes pour l'utilisateur.

I.3.1 Réflexivité

La réflexivité est la propriété d'un système qui est capable de raisonner à propos de lui-même [Maes et Nardi 1988]. Cette définition élémentaire, qui peut paraître obscure, introduit la notion de moyens d'analyse, de modification ou d'extension a posteriori du comportement initial du système. Ce concept est à la base même de celui de métaobjets, nous en donnons ici une définition.

I.3.1.1 Définition

Le terme **réflexif** est issu de la philosophie. Il est défini comme suit [Larousse 1972] :

> *Qui concerne la conscience se connaissant elle-même : la méthode réflexive remonte des conditions de la pensée à l'unité de la pensée.*

Le terme de **réflexivité** a, quant à lui, été introduit en logique pour désigner un moyen d'étendre les théories. Ensuite, il a été utilisé pour exprimer la relation entre la théorie et la méta-théorie dans les systèmes de raisonnement logique.

Plus récemment, la réflexivité a été adoptée comme un outil puissant et général des langages de programmation. On peut en donner la définition suivante [Maes et Nardi 1988] :

> *Un système informatique est dit réflexif lorsqu'il fait lui-même partie de son propre domaine. Plus précisément cela implique que (i) le système a une représentation interne de lui-même, et (ii) le système alterne parfois entre calcul «normal» à propos de son domaine externe et calcul «réflexif» à propos de lui-même.*

Ces définitions nécessitent des éclaircissements. On peut dire que le système réflexif contient une auto-représentation qui décrit la connaissance qu'il a de lui-même. Le système et son auto-représentation sont liés de manière causale : si l'un d'entre eux est modifié, l'autre l'est en conséquence. Un système réflexif peut donc s'auto-modifier en modifiant son auto-représentation, et son auto-représentation reste toujours cohérente avec son état réel.

On appelle **niveau de base** le système lui même. Le **méta-niveau** contient le modèle du niveau de base (l'auto-représentation du système) qu'il peut interpréter et modifier. Ces deux notions, illustrées par la figure 6, sont liées par deux processus d'interaction, la **réification** qui s'opère du niveau de base vers le méta-niveau lorsque le premier subit une modification devant être reflétée dans le second, et l'**intercession**, qui va du méta-niveau vers le niveau de base lorsqu'une modification du premier doit s'opérer sur le dernier.

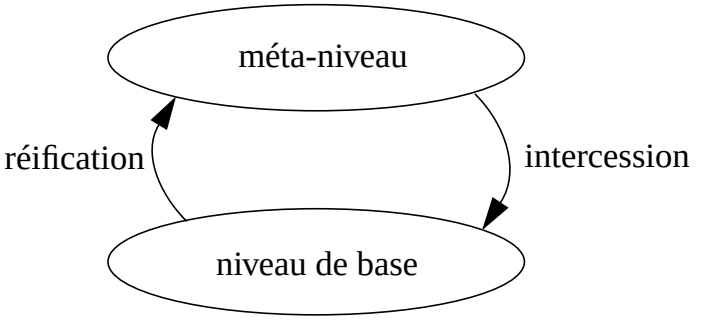

Figure 6 - Réflexivité : définitions

I.3.1.2 Niveaux multiples de réflexivité

Selon le système considéré, les propriétés apportées par la réflexivité peuvent être très différentes. Nous donnons ici plusieurs exemples de réflexivité afin d'illustrer les possibilités qu'offre cette notion dans le cadre général des systèmes informatiques.

I.3.1.2.1 Réflexivité du langage

Historiquement, la réflexivité a été introduite pour modéliser la représentation de la connaissance dans des systèmes d'intelligence artificielle, systèmes experts, etc. Ces différents systèmes raisonnent à partir d'une base de connaissance qu'ils peuvent enrichir de leurs propres déductions. On peut voir un tel système comme un programme interprété qui s'auto-modifie au fur et à mesure qu'il génère de nouvelles règles.

De nos jours, on peut classifier les langages de programmation en deux grandes catégories, les langages qui sont **interprétés** et ceux qui sont **compilés**. En effet cette différence est très importante en ce qui concerne la réflexivité.

Un programme écrit dans un langage interprété, comme en LISP, en Smalltalk ou encore en Java, est interprété par un programme spécifique appelé interpréteur. L'interpréteur évalue à l'exécution le programme source et exécute, en fonction de cette évaluation, son propre code machine. La réflexivité d'un tel langage permet donc au programme de s'auto-modifier à l'exécution permettant une très grande adaptabilité du programme à son environnement. Ces propriétés sont néanmoins obtenues au prix d'une certaine complexité.

Certains de ces langages, comme Perl [Dominus 1998], poussent cette notion de réflexivité très loin : l'interpréteur est considéré comme un programme à part entière qui peut lui-aussi être modifié par le programme qu'il interprète.

Cette approche de la réflexivité hérite tout autant des avantages que des inconvénients des langages interprétés. La méthode est très flexible car tout est contrôlable à l'exécution du programme, en revanche cette dynamicité se paye en termes de performances; en effet, l'interprétation d'un programme est très coûteuse en temps processeur.

A l'inverse des langages interprétés, les programmes écrits dans un langage compilé sont traduits en langage machine avant leur exécution. Le programme qui effectue cette traduction est appelé compilateur. De même que les langages interprétés réflexifs s'appuient sur un interpréteur réflexif, les langages compilés réflexifs s'appuient sur un compilateur réflexif. Celui-ci permet au méta-niveau de contrôler la façon dont le programme est compilé. Le méta-niveau peut alors être vu comme un ensemble de règles de traduction du programme source. On parle alors de compilateur ouvert, car le compilateur est programmable ; le méta-niveau est appelé méta-programme, il est responsable de la compilation du niveau de base, le programme source. La plupart des compilateurs réflexifs sont basés sur des langages orientés-objet, nous les détaillerons donc dans la section I.3.2.

I.3.1.2.2 Système-support d'exécution-middleware

Au-delà du langage, un système peut être réflexif en terme d'environnement d'exécution. L'environnement d'exécution d'un système, que ce soit le système d'exploitation, le support du langage, ou encore un middleware, peut être considéré comme réflexif s'il fournit les moyens de contrôler son propre comportement à un niveau supérieur (méta), en offrant donc un mécanisme de réification et d'intercession.

Le fait qu'un environnement d'exécution soit observable et contrôlable, grâce respectivement à la réification et à l'intercession, l'ouvre au monde extérieur et le rend donc extensible. A partir de là, toute sorte de méta-programme est envisageable. Dans [Salles *et al.* 1999], par exemple, la notion de micro-noyau réflexif est introduite. Les propriétés réflexives de tels noyaux sont utilisées pour la réalisation de mécanismes d'encapsulation (*wrappers*). Ces *wrappers*, qui encapsulent le noyau, vérifient certains prédicats pendant l'exécution. Ils permettent notamment d'améliorer la couverture des mécanismes de détection d'erreur et d'adjoindre des mécanismes de confinement pour des micro-noyaux du commerce.

Un autre exemple d'architecture réflexive est celle d'ABCL/R2 [Honda et Tokoro 1992]. Dans celle-ci, les propriétés réflexives sont utilisées pour contrôler le déroulement des applications dans un contexte temps-réel. Le contrôle des échéances des tâches est effectué au méta-niveau, de façon indépendante des tâches elles-mêmes; le méta-niveau gère une horloge et une liste de bornes temporelles d'exécution (démarrage et terminaison des tâches).

I.3.2 Protocoles à métaobjets

La réflexivité peut rester une notion très abstraite et confuse si elle n'est pas mise en œuvre de manière disciplinée et s'il n'existe pas de séparation claire entre niveau de base et méta-niveau. Cette confusion existe en Smalltalk-80, où une classe représente en même temps le niveau de base et le méta-niveau, ce qui nuit considérablement à la lisibilité des programmes.

Dans un modèle objet, la réflexivité est généralement vue à travers un protocole à métaobjets. Le **métaobjet** correspond au méta-niveau de l'objet, qui, pour sa part, représente le niveau de base. Le métaobjet peut donc observer l'objet par le biais de la réification. Il peut aussi contrôler l'objet grâce à l'intercession. Comme nous l'avons vu précédemment, dans un modèle à objets, différents objets interagissent par appel de méthodes. Ce principe s'applique de la même manière entre objets et métaobjets. On appelle l'ensemble des méthodes et des interactions qui régissent la réification et l'intercession entre objet et métaobjet, le **protocole à métaobjets**.

Puisque le concept d'objet diffère d'un modèle à l'autre, les informations relatives aux objets susceptibles d'être réifiées seront donc différentes ; c'est pourquoi il n'existe pas de protocole à métaobjets générique. Nous illustrons donc ce concept à travers deux exemples typiques de protocoles à métaobjets, puis nous décrivons au Chapitre II le protocole que nous avons conçu pour répondre à nos besoins.

Il existe une différence fondamentale au niveau du concept de **métaclasse** entre les protocoles à métaobjets à l'exécution ou à la compilation. Les premiers considèrent un métaobjet comme une instance de métaclasse alors que les seconds définissent la classe comme instance d'une métaclasse. Cette différence qui peut paraître troublante est néanmoins très logique. A l'exécution, l'entité la plus importante est l'objet ; le méta-niveau de celui-ci est le métaobjet qui est défini par une métaclasse. En revanche, à la compilation, seules les classes sont accessibles, les objets n'étant eux pas encore instanciés, le méta-niveau correspondant à la classe est donc une métaclasse.

I.3.2.1 Protocoles à métaobjets à l'exécution

La référence en matière de protocoles à métaobjets est sans aucun doute celui de CLOS. CLOS [Kleene 1989] est une extension orientée-objet de LISP dont le modèle à objets est basé sur les cinq concepts fondamentaux suivants : les classes, les attributs, les méthodes, les fonctions génériques et enfin la combinaison de méthodes. A chacun de ces concepts, CLOS associe un métaobjet différent qui contient les informations réflexives. Le protocole à métaobjets de CLOS fait l'objet d'un ouvrage complet [Kiczales *et al.* 1991] où la richesse et la complexité de celui-ci sont présentées en détail.

A titre illustratif, nous prendrons comme exemple de protocole à métaobjets à l'exécution celui proposé par Open-C++ v1 [Chiba et Masuda 1993] ; il a notamment été utilisé dans le développement du système FRIENDS [Fabre et Pérennou 1998] illustrant l'apport de l'utilisation de métaobjets dans un système distribué tolérant les fautes. Les qualités apportées par cette utilisation sont, par exemple, une très bonne séparation des concepts entre application et tolérance aux fautes ou sécurité, et une grande facilité de composition des différents mécanismes implémentés dans des métaobjets distincts et indépendants.

Le protocole à métaobjets d'Open C++ v1 est beaucoup plus simple que celui de CLOS. De plus, il a le mérite d'être appliqué à un langage couramment utilisé, C++. D'un point de vue pratique, un compilateur spécifique est utilisé pour lier classes et métaclasses. En effet, toutes les instances d'une même classe auront des métaobjets de la même métaclasse. L'utilisateur devra donc établir les liaisons causales entre classes et métaclasses ; c'est à dire déclarer la métaclasse qu'il désire associer à chaque classe.

Les informations du modèle objet de C++ qui sont réifiées par OpenC++ sont : la création ou la destruction d'un objet, l'appel d'une méthode, ainsi que la lecture ou l'écriture d'un attribut. Chaque évènement de ce type aura pour conséquence l'appel à une méthode du métaobjet. On dira que l'évènement est intercepté par le métaobjet, comme on peut le voir sur la figure 7.

La métaclasse *MetaObj* permet de définir des métaobjets neutres. Un métaobjet neutre ne fait qu'intercepter des évènements (création, invocation, etc.) et effectuer les actions correspondantes au niveau de base. Les métaobjets de cette classe ne modifient donc pas le comportement des objets : lors de la réification de la création

d'un objet, le métaobjet crée simplement l'objet ; lorsqu'une écriture d'attribut est réifiée, cette écriture est effectuée. Pour modifier ce comportement par défaut, l'utilisateur doit définir de nouvelles métaclasses. Celles-ci hériteront de la classe *MetaObj* et surchargeront certaines méthodes afin de réaliser les modifications voulues. Par exemple, pour rendre les objets persistants, il suffira au métaobjet de sauvegarder la valeur des attributs après chaque écriture et de relire ces valeurs à la prochaine création de l'objet.

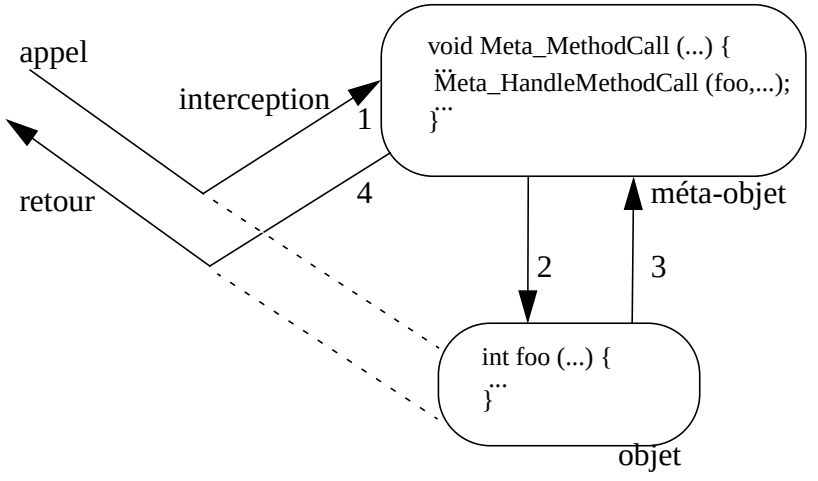

Figure 7 - Invocation de méthode avec Open C++ v.1

Comme on le voit sur la figure 8, les méthodes sont identifiées par un entier nommé *m_id* et les attributs le sont de la même façon par *a_id*. Les valeurs des attributs ou des paramètres de méthodes sont encapsulées dans une structure appelée *ArgPack*. Cela permet à l'interface des métaclasses de rester indépendantes du nombre et du type des arguments et attributs.

```
class MetaObj {
public:
 void Meta_MethodCall (int m_id, ArgPack& args, ArgPack& reply);
 void Meta_Read (int a_id, ArgPack& value)
 void Meta_Assign (int a_id, ArgPack& value);
 void Meta_StartUp ();
 void Meta_CleanUp ();
protected:
 void Meta_HandleMethodCall (int m_id, ArgPack& args, ArgPac& reply);
 void Meta_HandleRead (int a_id, ArgPack& value);
 void Meta_HandleAssign (int a_id, ArgPack& value);
};
```

Figure 8 - Interface de la classe MetaObj

Les méthodes *Meta_HandleMethodCall*, *Meta_HandleRead* et *Meta_HandleAssign* permettent au métaobjet d'effectuer les opérations d'appel de méthode et de lecture et écriture d'attribut sur l'objet. Elles implémentent donc le comportement par défaut de ces opérations et sont les seuls moyens d'accès à l'objet pour le métaobjet.

Les méthodes *Meta_StartUp* et *Meta_CleanUp* sont invoquées respectivement après la création et avant la destruction de l'objet, alors que *Meta_MethodCall*, *Meta_Read* et *Meta_Assign* obtiennent le contrôle avant et après chaque invocation de méthode, lecture ou écriture d'attribut. Cette asymétrie, gênante d'un point de vue conceptuel, vient du fait qu'objet et métaobjet sont à l'exécution une seule et même entité, le métaobjet est donc créé et détruit en même temps que l'objet.

I.3.2.2 Protocoles à métaobjets à la compilation

On reproche parfois aux protocoles à métaobjets un manque d'efficacité dû notamment aux nombreuses indirections nécessaires entre objets et métaobjets. Open-C++ v2 [Chiba 1995] permet d'utiliser la réflexivité à la compilation et non à l'exécution pour pallier ce manque d'efficacité. Dans ce cas, une métaclasse est responsable de la compilation d'une classe et peut donc personnaliser le code produit par le compilateur (cf. figure 9).

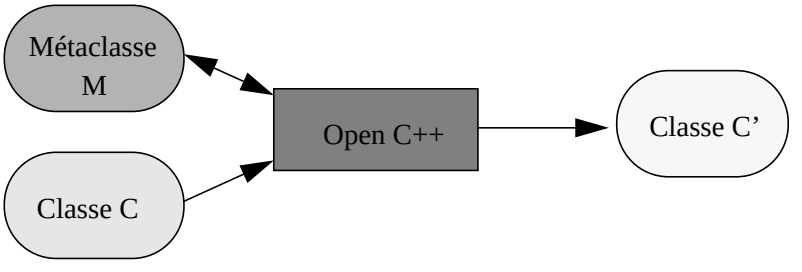

Figure 9 - Principe de fonctionnement d'Open C++ v2

Outre l'efficacité, un des plus gros avantages d'une telle approche est le volume des informations disponibles au méta-niveau. Avec Open-C++ v1 seules les informations création, destruction, invocation, lecture et écriture étaient disponibles. Avec Open-C++ v2 les métaclasses ont accès à toutes les notions définies par le modèle à objets du langage C++. Celles-ci incluent notamment l'héritage multiple, la composition d'objets, le type des attributs, l'utilisation de *templates*, etc. Il est clair que la richesse des méta-informations fournies se paye en terme de complexité du protocole à métaobjets ; nous essayons toutefois ici de donner une idée de ce qu'il est possible de faire avec un tel protocole.

On peut voir les métaclasses comme des programmes capables de personnaliser le comportement d'un compilateur, c'est pourquoi on parle de programmes de méta-niveau. Le compilateur est dit ouvert car il délègue à la métaclasse les décisions à prendre sur la traduction du programme source. Il est donc facilement configurable : on modifie son comportement en changeant de métaclasse.

Le compilateur analyse le code source d'une classe, effectue une analyse syntaxique et grammaticale puis construit une représentation arborescente du code source. Chaque branche représente un élément particulier de la définition de la classe

actuellement compilée : la représentation arborescente complète correspond à la classe, certaines branches représentent les attributs, d'autres représentent les méthodes, etc. A partir de l'analyse de cette représentation arborescente, le compilateur réifie à la métaclasse divers types de métainformation : déclaration d'une méthode, instanciation de la classe, etc. La métaclasse peut donc réagir à ces informations en produisant du code qui se substituera au code original, cela constitue le mécanisme d'intercession de ce type de protocoles à métaobjets. La métaclasse définit donc des règles de traduction de la classe de base en utilisant le protocole à métaobjets en lecture (réification de la classe de base) ou en écriture (modification par intercession).

Le protocole à métaobjets d'Open-C++ v2 est basé sur plusieurs classes de métaobjets :

- Les métaobjets *Ptree* qui fournissent une représentation sous forme arborescente du code source du programme.
- Les métaobjets *Environment* qui représentent les liaisons entre les noms de variables ou d'attributs et leurs types.
- Les métaobjets *TypeInfo* qui représentent les types tels qu'ils apparaissent dans le programme. Ils incluent les types dérivés tels que pointeurs, références aussi bien que les types de base et les classes.
- Les métaobjets *Member* qui représentent méthodes et attributs. Ils permettent d'obtenir des informations sur les types des paramètres d'entrée et de retour, etc.
- Les métaobjets *Class* qui représentent la définition des classes et qui contrôlent la traduction de celles-ci.

L'interface de la classe *Class* est très fournie. Elle permet en premier lieu de faire de l'introspection sur la classe de base : obtention du nom, de la hiérarchie d'héritage, de la liste des attributs et méthodes ainsi que leurs métaobjets *Member*. D'autre part, elle permet de réifier des modifications portant sur la hiérarchie d'héritage, la liste des attributs et méthodes ainsi que le code associé. Enfin elle permet de personnaliser l'initialisation des objets, les affectations, les opérateurs binaires, unaires, pré et post-fixés, les appels de méthodes, les création et destructions d'objets, les lectures et écritures d'attributs, etc.

L'utilisateur peut définir sa propre métaclasse en héritant de la classe *Class*. Il redéfinit alors les méthodes correspondant aux modifications qu'il veut apporter. S'il veut par exemple rendre une classe persistante il devra alors réécrire les méthodes *TranslateMemberWrite* et *TranslateMemberFunction* afin de sauvegarder les attributs lors de leur écriture et recharger leur valeur à la création de l'objet.

OpenJava [Tatsubori 1999], quant à lui, repose sur le même modèle, mais applique le concept de réflexivité à la compilation au langage Java. Le parallèle entre ces deux outils nous sera utile dans les chapitres suivants.

I.4 Conclusion

Nous avons, dans ce chapitre, introduit les concepts de la programmation orientée-objet : les classes et leurs instances, les objets, les notions plus avancées d'héritage, de délégation et de polymorphisme. En effet, les diverses technologies orientées-objet se sont révélées très utiles pour le développement des systèmes distribués, de leur conception à leur implémentation.

Les systèmes distribués sont de nos jours au cœur des préoccupations. En effet, Internet et les diverses technologies qui en ont émergé, comme Java, ont démocratisé l'utilisation des réseaux informatiques. Ces évolutions technologiques ont mené à l'utilisation d'un nouveau modèle, ou paradigme, celui des systèmes distribués : différentes machines, interconnectées, qui coopèrent dans un but commun, ou qui sont en compétition pour l'accès à des ressources communes. Ce nouveau type de système offre plus de souplesse et de puissance aux applications. Mais dans le même temps, de par leur hétérogénéité, ils sont bien plus complexes à gérer. Afin de faciliter leur utilisation, plusieurs outils et normes ont alors fait leur apparition : Java, DCOM, CORBA. CORBA est le plus générique d'entre eux : cette norme est définie de telle façon que les applications peuvent se baser sur elle seule, et non plus sur la plate-forme constituée du matériel et du système d'exploitation. CORBA est devenu en quelques années l'outil idéal pour le développement de systèmes distribués à objets.

Une autre évolution importante des technologies orientées-objet a été constituée par l'utilisation de la notion de réflexivité. La réflexivité est la propriété d'un système de raisonner et d'agir sur lui-même. L'application de ce concept aux technologies orientées-objet mène à la notion de protocole à métaobjets. Un métaobjet peut contrôler et agir sur un objet en utilisant le protocole à métaobjets. Nous avons présenté en section I.3 diverses utilisations de ces concepts, qu'ils soient utilisés au moment de la compilation ou de l'exécution de l'application. L'avantage majeur des protocoles à métaobjets est de permettre le développement séparé et indépendant de l'application et de mécanismes non-fonctionnels, de tolérance aux fautes par exemple pour ce qui nous concerne. Des travaux précédents ont porté sur l'utilisation de protocoles à métaobjets COTS (Component-Off-The-Shelf/Composant sur étagère) pour la tolérance aux fautes d'applications distribuées et ont montré les limites de ces protocoles en terme de contrôle des objets. Nous proposons, dans cette thèse, une nouvelle approche qui permet de repousser ces limites, et qui consiste à utiliser la réflexivité à la compilation pour générer un protocole à métaobjets spécifique à la tolérance aux fautes des objets CORBA.

CHAPITRE II

TOLÉRANCE AUX FAUTES DANS LES SYSTÈMES DISTRIBUÉS À OBJETS

Un des objectifs du protocole à métaobjets pour les objets CORBA que nous proposons dans cette thèse est de permettre l'implémentation de mécanismes de tolérance aux fautes, et ce, de manière à la fois souple et efficace. Notre propos n'est pas de concevoir de nouveaux mécanismes de tolérance aux fautes mais de définir un environnement d'utilisation de tels mécanismes sur une plate-forme CORBA. Nous nous intéressons donc ici au support et au cadre d'intégration de ces mécanismes ainsi qu'à leur mise à disposition de l'utilisateur.

Dans un premier temps, nous présentons brièvement les concepts de base de la tolérance aux fautes dans les systèmes distribués ; nous donnons tout d'abord quelques définitions, puis les modèles et hypothèses de fautes. Dans la deuxième section, nous cherchons à évaluer les différentes approches existantes au problème de la tolérance aux fautes dans les systèmes distribués à objet. Nous les présentons et les comparons en fonction de critères que nous aurons préalablement définis. Enfin, cette évaluation nous permet de situer notre approche par rapport à celles présentées.

II.1 Introduction et concepts de base

Dans cette section, nous abordons les concepts de sûreté de fonctionnement et de tolérance aux fautes. Après avoir donné quelques définitions, nous présentons un modèle de système distribué pour la tolérance aux fautes et les hypothèses sous-jacentes.

II.1.1 Définitions

La **sûreté de fonctionnement** d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Laprie *et al.* 1996]. Le *service* délivré par un système est son comportement tel que perçu par son ou ses utilisateurs ; un *utilisateur* est un autre système (humain ou physique) qui inter-agit avec le système considéré.

En fonction des applications auxquelles est destiné le système, les besoins en sûreté de fonctionnement sont différents. On peut, par conséquent, voir la sûreté de fonctionnement selon différentes facettes, selon différents **attributs** :

- la *disponibilité* évalue la capacité d'être prêt à rendre le service ;
- la *fiabilité* cherche la continuité du service ;
- la *sécurité-innocuité* est définie par la non-occurrence de conséquences catastrophiques pour l'environnement ;
- la *confidentialité* recherche la non-occurrence de divulgations non-autorisées de l'information ;
- l'*intégrité* permet la non-occurrence d'altérations non appropriées du système ;
- la *maintenabilité* est l'aptitude aux réparations et aux évolutions.

Dans le cadre de cette thèse, nous nous intéressons plus particulièrement à la mise en place de mécanismes permettant d'assurer la disponibilité et la fiabilité du système.

Parmi les **entraves** à la sûreté de fonctionnement, on distingue les fautes, les erreurs et les défaillances.

- Une *défaillance* du système survient lorsque le comportement du système dévie de sa fonction, c'est à dire que le service rendu diverge de celui qui est attendu.
- Une *erreur* est la partie de l'état du système susceptible d'entraîner la défaillance.
- Une *faute* est la cause adjugée ou supposée d'une erreur.

Dans la pratique, fautes, erreurs et défaillances peuvent s'enchaîner sans fin comme suit :

$$... \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow \text{erreur} \rightarrow \text{défaillance} \rightarrow \text{faute} \rightarrow ...$$

Plus précisément, la défaillance d'un composant est considérée comme une faute pour le système qui l'englobe. Cette faute peut entraîner une erreur, auquel cas la faute est active. Une erreur reste latente tant qu'elle n'a pas été reconnue comme telle. Elle peut créer de nouvelles erreurs par propagation. La défaillance survient lorsque le service délivré est affecté.

La *tolérance aux fautes* est un des **moyens** de la sûreté de fonctionnement qui cherche à permettre au système de remplir sa fonction en dépit de la présence de fautes. Les autres moyens sont la *prévention des fautes*, qui tend à empêcher l'introduction de fautes dans le système, l'*élimination de fautes*, qui veut réduire la présence des fautes en termes de nombre ou de sévérité et, enfin, la *prévision de fautes*, qui cherche à estimer la présence et les conséquences des fautes.

II.1.2 Modèle pour la tolérance aux fautes

Pour définir ou modéliser un système distribué tolérant aux fautes, deux types d'information sont particulièrement importants : le modèle de communication sous-jacent et les modes de défaillance du système. Le modèle de communication dépend fortement de l'hypothèse de synchronisme que l'on peut faire sur le système. Les modes de défaillance décrivent les hypothèses de fautes que l'on peut faire sur le système et influent sur le type de mécanismes de tolérance aux fautes à mettre en place afin d'augmenter sa sûreté de fonctionnement.

II.1.2.1 Systèmes synchrones et systèmes asynchrones

Les communications ont un rôle prépondérant dans les systèmes distribués où différentes entités doivent inter-agir. Leur coopération repose donc sur un ou plusieurs protocoles, sur les propriétés qu'ils offrent et sur des hypothèses sous-jacentes. La première -et la plus importante- des caractéristiques d'un système distribué tolérant les fautes est son modèle temporel.

Le modèle temporel le plus simple est celui d'un système **asynchrone**, dans lequel on ne fait aucune hypothèse sur les temps d'exécution et de livraison des messages. Un message envoyé par un nœud non-défaillant à un autre nœud, à travers un canal non-défaillant, sera éventuellement reçu et traité, mais on ne peut garantir de borne temporelle [Powell 1998]. Les algorithmes conçus pour un tel modèle sont très génériques puisqu'ils ne se fondent sur aucune hypothèse restrictive.

A l'opposée du modèle asynchrone, on trouve le modèle **synchrone**. Dans un système synchrone, chaque message envoyé par un nœud non-défaillant à un autre nœud est reçu et traité en un temps borné. Dans la pratique, pour faire cette hypothèse, il est nécessaire :

- d'utiliser des techniques de contrôle de flux et un ordonnanceur temps-réel «dur» afin de borner les temps d'exécution et de livraison des messages,
- et de faire l'hypothèse d'une borne supérieure du nombre de défaillances qui peuvent survenir par unité de temps [Fetzer et Cristian 1997].

C'est un modèle puissant et approprié aux applications critiques qui nécessitent des garanties temps-réel en présence de fautes. Toutefois les hypothèses qu'il requiert doivent être justifiées par l'utilisation de canaux de communications et d'un système d'exploitation appropriés.

De telles hypothèses permettent d'employer des protocoles plus simples ou plus efficaces qu'avec un système asynchrone et certains protocoles sont impossibles à implémenter sans ces hypothèses [Fischer *et al.* 1985]. Les protocoles d'élection en sont un bon exemple : il s'agit pour un groupe de n processus d'élire un leader en utilisant un protocole de communication de groupe sûr. Résoudre ce problème avec un système asynchrone nécessite l'émission de n messages alors qu'une seule émission suffit à un système synchrone [Schneider 1993]. En effet, un protocole simple

pour système synchrone est d'élire comme *leader* l'émetteur du premier message reçu (le protocole de communication de groupe assurant l'ordonancement des messages).

Avec les grands réseaux actuels, comme Internet par exemple, on remarque rapidement que les temps de communication varient énormément et ne peuvent être bornés. Malgré cela, il est courant, dans la pratique, de les considérer comme synchrones et d'utiliser des temporisations (*time-out)* pour détecter les nœuds ayant défailli. Dans ce cas, la valeur des temporisations est sur-dimensionnée afin que la probabilité d'une fausse détection soit faible et proportionnée au niveau nécessaire de sûreté de fonctionnement du système considéré.

L'idéal pour modéliser de tels réseaux serait un modèle intermédiaire entre synchrone et asynchrone.

- Une approche prometteuse est celle du modèle **asynchrone temporisé**, qui fait l'hypothèse que les nœuds non-défaillants ont une horloge locale, et que l'on peut borner la dérive de celle-ci par rapport au temps «réel», c'est-à-dire, par rapport aux horloges des autres nœuds du système. En utilisant ces horloges pour assigner des *estampilles temporelles (timestamp)* aux messages, il est possible de calculer a posteriori les limites supérieures des délais de transfert subits par les messages. On peut alors identifier (également a posteriori) des périodes de temps où le système se comporte de façon synchrone et où l'on peut garantir certaines propriétés. Hors de ces périodes, on suspecte le système de se comporter de façon asynchrone et le fonctionnement est alors dégradé mais néanmoins sûr. Ce modèle est particulièrement utile pour implémenter des systèmes distribués sûrs en présence de défaillances [Fetzer et Cristian 1997]. Il a, en particulier, été utilisé dans des systèmes automatiques de transport [Essame 1998].
- Un autre modèle intermédiaire est permis par l'emploi de **detecteurs de défaillances non-sûrs** [Chandra et Toueg 1996]. Dans un système asynchrone, il est impossible de distinguer un nœud lent d'un processus qui a défailli [Fischer *et al.* 1985]. Pour pallier à cette impossibilité, on doit s'autoriser à suspecter un nœud non-défaillant d'avoir défailli. Sous certaines hypothèse, on peut alors implémenter des protocoles de consensus pour des défaillances par arrêt [Chandra et Toueg 1996] ou pour des défaillances temporelles [Hurfin *et al.* 1998]. Le protocole de consensus est une brique de base des mécanismes de tolérance aux fautes.

Dans cette thèse, nous nous intéressons particulièrement à améliorer la disponibilité et la fiabilité d'applications CORBA et nous ne considérons pas la sécurité-inocuité. La norme CORBA utilise un modèle de système asynchrone, nous utiliserons néanmoins l'approche décrite ci-dessus qui consiste à employer des temporisations pour la détection des noeuds ayant défaillis.

II.1.2.2 Modes de défaillance et hypothèses

On dit qu'un composant du système est correct si le service qu'il rend est conforme à sa fonction. La défaillance d'un composant survient lorsque ce composant dévie de l'accomplissement de sa fonction. Dans l'absolu, un système distribué peut défaillir de différentes façons. De nombreuses classifications de défaillances ont été données [Powell 1992], [Cristian *et al.* 1995], [Laprie *et al.* 1996] sur lesquelles il n'y a pas de consensus. Nous utiliserons ici la classification issue de [Powell 1992] car elle a le mérite de prendre en compte le fait qu'un service puisse être rendu à plusieurs clients et elle s'applique plus aisément aux systèmes distribués.

Le service rendu par un système peut défaillir schématiquement suivant deux axes : l'axe des valeurs (la valeur rendue en réponse à une requête n'est pas correcte), et l'axe du temps (la réponse ne se fait pas dans l'intervalle de temps spécifié). Sur chacun de ces deux axes, on distingue ensuite plusieurs types d'erreurs.

Sur l'axe des valeurs, on peut distinguer :

- Les **erreurs arbitraires de valeur**. La valeur rendue par le service ne fait pas partie de l'ensemble des réponses spécifié pour ce service. Cette définition est la plus générale, l'adjectif «arbitraire» souligne le fait que la définition ne pose aucune restriction sur la valeur erronée.
- Les **erreurs de valeur hors-code** : la valeur rendue par le service est hors du code qui spécifie l'ensemble des réponses possibles pour tous des services. Ces erreurs sont faciles à détecter, on peut utiliser un code détecteur d'erreur à cette fin.

Dans la figure 10, les deux ensembles définis ci-dessus sont illustrés. Dans cet exemple, le code spécifié pour l'ensemble des services est l'ensemble des mots de quatre lettres. Deux fonctions sont définies par un même service, la premiere répond à chaque requête «pile» ou «face», la seconde fonction retourne une couleur parmi l'ensemble {bleu, vert, noir et brun}. Chaque réponse possible de ces deux services est un mot de quatre lettres, invariant qu'il est facile de vérifier. Un exemple de défaillance arbitraire serait que la seconde fonction renvoie la valeur «gris», qui est bien un mot de quatre lettres, et, qui plus est, une couleur. Une erreur hors-code serait, par exemple, que cette même fonction retourne «rouge», qui contient cinq lettres et par conséquent est «hors-code». On voit bien qu'une erreur hors-code est facile à détecter, il suffit de compter dans ce cas le nombre de caractères de la réponse. En revanche, pour détecter une erreur arbitraire, une analyse sémantique est nécessaire.

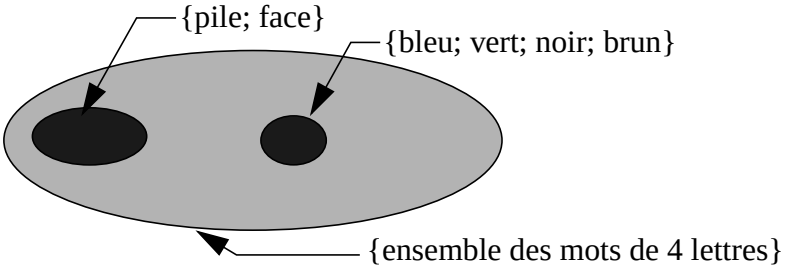

Figure 10 - Erreurs en valeurs

Sur l'axe temporel (illustré par la figure 11), on distingue plusieurs types d'erreurs.

- Les **erreurs temporelles arbitraires** : il s'agit du cas le plus général où la réponse n'est pas rendue dans l'intervalle de temps spécifié.
- Les **erreurs par omission** : la réponse du service à une requête n'est pas et ne sera jamais rendue.
- Les **erreurs par retard** : la réponse au service est rendue en retard par rapport à la spécification du service.
- Les **erreurs par avance** : la réponse est rendue en avance par rapport au service attendu.

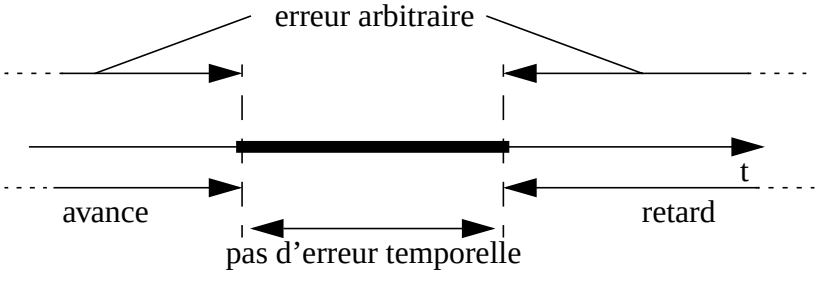

Figure 11 - Erreurs temporelles

Les différents types d'erreurs en valeur et temporelles se combinent deux à deux. Les couples ainsi formés sont classés par une relation d'ordre partiel. Cette relation permet de dire, par exemple, qu'un système qui tolère les erreurs arbitraires en valeur et en temps, tolère également tout autre type d'erreur. Certains des couples définissent des classes d'erreurs remarquables, c'est le cas notamment du couple (pas d'erreur en valeur; erreur permanente par omission), qui définit le comportement d'un composant à **silence sur défaillance**.

Enfin, dans le cas ou le service est rendu à plusieurs utilisateurs, de nouveaux types d'erreurs peuvent être définis. On dit que le service est **cohérent** si chacun des utilisateurs reçoit la même réponse ; le service est correct si cette réponse n'est pas erronée. On notera que ce modèle précise également une condition de **cohérence temporelle** : la différence de temps entre chacune des réponses faites aux différents

utilisateurs est bornée (les instants de délivrance étant pris deux à deux). Evidemment cette condition ne peut être utilisée que lorsque le système est synchrone ou asynchrone temporisé.

- Les **erreurs d'incohérence** (en valeur ou en temps) : les conditions de cohérence ne sont pas vérifiées, soit que chacun des clients ne reçoit pas la même réponse, soit que certains d'entre eux reçoivent cette réponse hors de l'intervalle de temps spécifié.
- Les **erreurs cohérentes en valeur** : la valeur rendue à chaque utilisateur est la même, mais elle est incorrecte.
- Les **erreurs temporelles cohérentes** : le service est rendu en dehors de l'intervalle de temps spécifié mais la condition de cohérence temporelle est vérifiée.
- Les **erreurs semi-cohérentes en valeur** : un groupe d'utilisateur reçoit la même réponse erronée (arbitraire) et les autres reçoivent une réponse hors-code.

II.2 Les différentes approches

Nous présentons, dans cette section, les différentes façons d'appréhender la tolérance aux fautes dans les systèmes répartis que l'on peut rencontrer dans les systèmes actuels. Comme nous le verrons, des différences fondamentales existent entre tous ces systèmes. Elles sont essentiellement dues à la couche du système dans laquelle sont intégrés les mécanismes de tolérance aux fautes, ainsi qu'à la technique d'intégration utilisée entre l'application et les mécanismes.

Après avoir donné une brève présentation de ces systèmes, nous cherchons à évaluer leurs caractéristiques sous l'angle de plusieurs critères. Ces critères nous permettent alors de les comparer.

Ces différentes approches peuvent être classifiées selon la couche du système dans laquelle sont intégrés les mécanismes de tolérance aux fautes. Comme on le voit sur la figure 12, on distingue :

- des approches de bas niveau :
  - les approches *système,* où les mécanismes sont intégrés dans les couches les plus basses, et qui ont accès à un certain nombre d'informations qui ne sont accessibles qu'à l'intérieur du système.
- des approches de niveau intermédiaire (*middleware*) :
  - les *librairies,* qui sont implémentées entre le système et l'application, ou plus précisément que l'utilisateur doit lier à son application. Elles utilisent les services du système pour fournir des briques de base que l'application peut utiliser pour implémenter ses propres mécanismes.
  - celles par *interception,* qui utilisent des mécanismes particuliers du système d'exploitation pour intercepter les évènements générés par l'ORB utiles aux mécanismes.
  - les approches par *intégration,* qui, comme leur nom l'indique, intègrent les mécanismes de tolérance aux fautes à l'ORB.

- et des approches au niveau applicatif :
  - les *services*, qui implémentent certains mécanismes de tolérance aux fautes au-dessus du middleware, à charge de l'utilisateur de les appeler.
  - et enfin les systèmes *réflexifs* ou par *héritage*, qui se placent au niveau du *langage*. Ils utilisent des mécanismes du langage, réflexivité ou héritage afin de lier mécanismes et application.

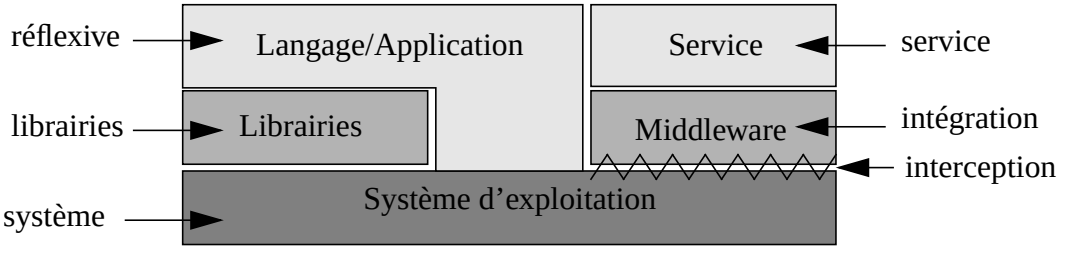

Figure 12 - Différentes couches pour la tolérance aux fautes

II.2.1 Niveau système

Les approches dites «système», proposent d'intégrer les mécanismes de tolérance aux fautes dans les couches basses du système distribué que sont le noyau ou le système d'exploitation. Le fait d'être au cœur du système permet à ces mécanismes d'accéder à des informations qui sont cachées aux niveaux d'abstraction plus élevés. Cette approche permet également de gérer d'éventuels périphériques spécifiques, comme les contrôleurs de communication à silence sur défaillance de Delta-4 par exemple.

Le système DELTA-4 [Powell 1991] est un exemple complet d'un système tolérant aux fautes qui, de certains points de vue, offre des similitudes vis-à-vis de CORBA. Il définit une architecture, certains aspects de sa conception, son implémentation et sa validation. L'objectif initial de ce projet est de considérer un système distribué constitué de stations de travail du commerce (COTS). Les mécanismes de tolérance aux fautes sont fondés sur l'utilisation d'objets et de messages. Le système de communication de groupe utilise des contrôleurs réseau à silence sur défaillance (NAC). L'implémentation des NACs repose sur l'utilisation de matériel auto-testable et de mécanismes d'exclusion mémoire. La figure 13 illustre cette architecture.

Sur cette base matérielle, de nombreux mécanismes de réplication ont été développés : sauvegarde sur support stable, réplication semi-active, réplication active avec vote à la source et à la destination, etc. Les protocoles inter-répliques de ces mécanismes sont intégrés à bas niveau, dans le système de communication de groupe.

Au dessus de ces services de bas niveau, DELTA-4 définit un environnement de développement et d'exécution. Cet environnement définit un langage de description d'interfaces, à la manière de l'IDL CORBA, qui permet à l'utilisateur d'utiliser le langage de programmation de son choix. Les unités de réplication sont les *capsules*, elles peuvent être comparées aux objets CORBA. La sélection de la stratégie de

réplication se fait de façon externe à l'application au moment de la configuration du système. Ainsi, la tolérance aux fautes est transparente pour l'utilisateur et les stratégies de réplication peuvent être choisies sans modification de l'application.

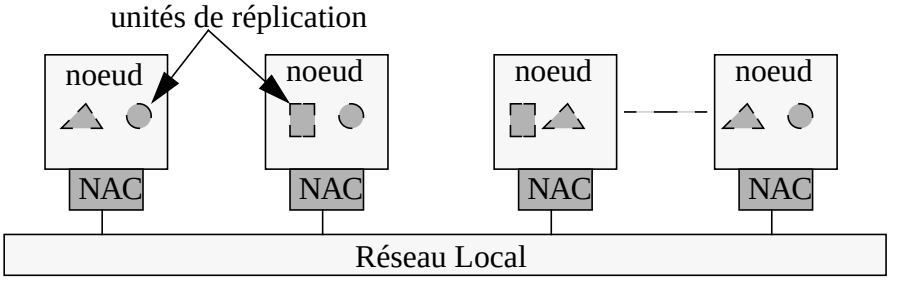

Figure 13 - L'architecture de Delta-4

Le modèle de programmation de DELTA-4 est donc relativement proche de celui de CORBA. L'utilisation d'un langage de description d'interfaces, d'unités de réplication en tant qu'objets, de réseaux de communications standards, font que l'implémentation d'un ORB au dessus de DELTA-4 ne serait pas difficile.

DELTA-4 définit un modèle à objets qui est ancré au cœur du système, tout comme le sont les mécanismes de tolérance aux fautes. Ce système offre donc transparence, séparation et configurabilité à l'utilisateur, mais ni réutilisation des mécanismes, ni leur composition. Son approche permet d'utiliser divers langages grâce au langage de description d'interfaces.

II.2.2 Les librairies

Les librairies de tolérance aux fautes, libckpt [Plank *et al.* 1995] et libft [Huang *et al.* 1995] par exemple, fournissent à l'utilisateur des mécanismes de tolérance aux fautes à reprise par recouvrement pour des processus Unix. Elles peuvent être utilisées dans un mode transparent où l'utilisateur n'a pas à se soucier du fonctionnement des mécanismes mais permettent également à celui-ci de configurer plus finement la façon avec laquelle ceux-ci sont menés. Dans ce dernier mode de fonctionnement, l'utilisateur peut également personnaliser la manière avec laquelle l'état de son application est obtenu.

Par conséquent, bien qu'initialement prévues pour la sauvegarde et la restauration de simples processus, ces librairies peuvent être utilisées avec des serveurs CORBA. En effet, l'utilisateur peut configurer les fonctions des librairies pour sauvegarder et restaurer correctement les pointeurs ou les références d'objets CORBA, ce qui n'est pas le cas par défaut. L'utilisateur doit également implémenter les mécanismes de réplication. En effet, ces librairies ne fournissent, a priori, que des mécanismes à base de sauvegarde sur support stable.

L'utilisation de telles librairies au sein d'applications distribuées à objets n'offre ainsi que peu d'avantages. Non seulement l'utilisateur doit les adapter pour qu'elles gèrent correctement les références entre objets et autres particularités de son système, mais il doit aussi implémenter les mécanismes de tolérance aux fautes au sein de son application.

II.2.3 Entre système et ORB : l'interception

Le système ETERNAL [Moser et Melliar-Smith 1997] propose d'implémenter des mécanismes de réplication au sein d'un ORB, de telle manière que l'utilisation de ces mécanismes soit transparente du point de vue de l'utilisateur. L'un des objectifs d'ETERNAL est de pouvoir utiliser n'importe quel ORB du commerce. En effet, l'approche repose sur l'utilisation du protocole Internet-Inter-ORB Protocol (IIOP) qui fait partie du standard CORBA. Le protocole IIOP utilise généralement les services TCP-IP du système. L'approche consiste à intercepter les messages TCP-IP qui transitent entre l'ORB et le système d'exploitation, ces messages sont déroutés vers un système de communication de groupe, ici TOTEM [Moser *et al.* 1996]. Ce mécanisme d'interception est donc très dépendant du système d'exploitation sous-jacent, ici Unix. L'architecture globale du système est donnée par la figure 14.

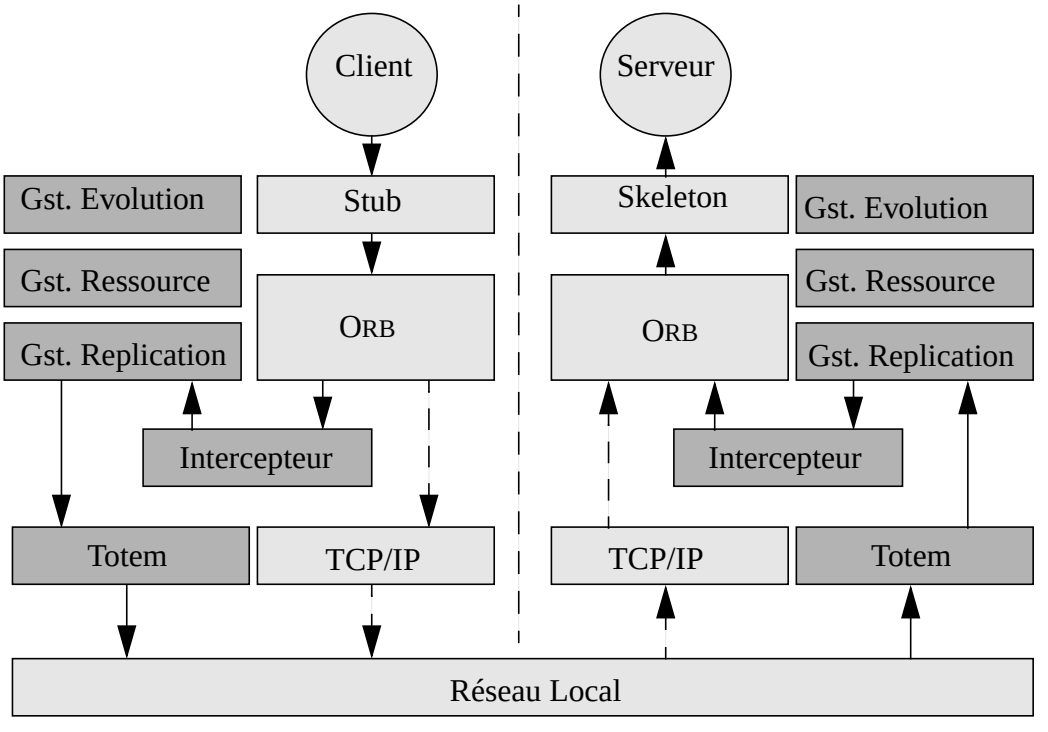

Figure 14 - Architecture du système Eternal

Le Gestionnaire de Réplication (*Replication Manager*) implémente les différentes stratégies de réplication en utilisant le service de communication de groupe fourni par TOTEM. Il reçoit les requêtes à partir de l'*intercepteur*, et envoie les messages

correspondant au groupe de répliques selon la stratégie choisie. Il gère la duplication des requêtes, les transferts d'états ainsi que les éventuels partitionnements du réseau. Le Gestionnaire de Ressources (*Resource Manager*) administre les domaines de réplication et leurs différentes ressources. Le Gestionnaire d'Evolution (*Evolution Manager)* est responsable de la création et de la destruction des répliques dans les phases de configuration ou de reprise du système.

On est en droit de se poser plusieurs questions à propos d'ETERNAL. En effet, les documents disponibles ne traitent pas de l'obtention de l'état des objets, les fonctions idoines sont sans doute à la charge de l'utilisateur. De plus, il semble que le Gestionnaire de Replication soit responsable de la configuration des mécanismes de tolérance aux fautes. Afin de personnaliser les mécanismes, on peut raisonnablement penser qu'il faille modifier celui-ci. De même, pour implémenter de nouveaux mécanismes, il est nécessaire d'implémenter un nouveau Gestionnaire de Replication. En revanche, les avantages de cette approche sont évidents : elle s'adapte à tout ORB du commerce si le système d'exploitation sous-jacent supporte le mécanisme d'interception, ce qui devrait être le cas au moins pour tous les systèmes Unix.

II.2.4 Intégration à l'ORB

Le système ELECTRA [Maffeis et Schmidt 1997] propose une approche de la tolérance aux fautes dans un ORB complètement différente de celle prise dans ETERNAL. En effet, cette architecture propose d'intégrer directement les mécanismes de tolérance aux fautes et le système de communication de groupe au sein de l'ORB. ELECTRA se compose donc d'un ORB spécifique, qui repose sur plusieurs couches logicielles comme on le voit sur la figure 15.

ELECTRA peut utiliser différents systèmes de communication de groupe : Isis, Horus ou encore Ensemble. Un adaptateur (*Adaptor*) différent est utilisé pour chacun de ces systèmes, il fournit une interface commune au dessus de ces différents systèmes. Ainsi, la machine virtuelle (*Virtual Machine*) peut utiliser les services de ces systèmes de façon indépendante. Cette machine virtuelle, ainsi que le système d'exploitation virtuel (*VOS*), permet de porter le système ELECTRA sous différents environnements. Ces deux entités fournissent des fonctions de communication, de gestion des tâches et d'autres services relatifs au système d'exploitation. Enfin, un ORB est implémenté au dessus de cette machine et de ce système d'exploitation virtuels (*DII, SII, ORB* et *BOA*). C'est un ORB qui respecte la norme CORBA hormis le fait que son adaptateur d'objets (BOA-Basic Object Adapter) permet, en plus des fonctionnalités standard, la gestion de groupes de répliques : création d'un groupe, ajout d'un nouveau membre, changements de vue, etc. Les objets de l'application doivent implémenter les méthodes *get_state* et *set_state* afin que le BOA puisse respectivement sauvegarder et restaurer leur état et ainsi participer à l'implémentation de différents mécanismes de réplication.

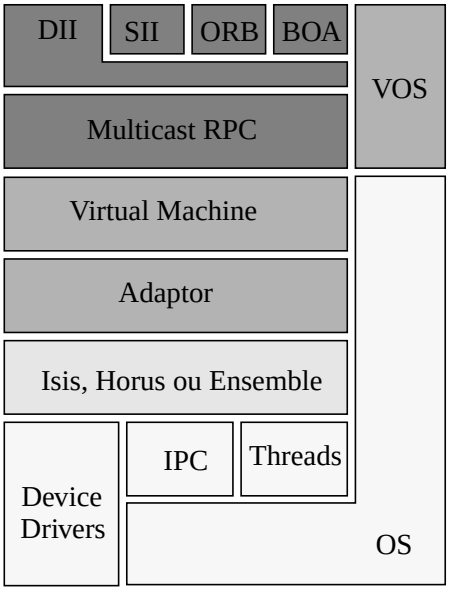

Figure 15 - Architecture du système Electra

Cette approche semble relativement portable grâce aux abstractions utilisées : machine et système d'exploitation virtuels. Néanmoins, elle requiert l'utilisation d'un ORB spécifique. De plus, les mécanismes de tolérance aux fautes se trouvent divisés dans différentes entités du système : le BOA gère l'état des objets, mais ces derniers sont impliqués dans la gestion des groupes. Les mécanismes de tolérance aux fautes ne sont donc pas transparents pour l'utilisateur de ce système.

II.2.5 Services pour la tolérance aux fautes

L'architecture CORBA encourage à développer les outils et mécanismes non-fonctionnels ou récurrents sous la forme de services (cf I.2.2.3). Ces services implémentent chacun une interface claire, définie en IDL et ainsi peuvent être facilement utilisés par les développeurs. Le projet OPENDREAMS [Felber *et al.* 1996] se propose d'implémenter un système de communication de groupe sous la forme d'un service (OGS), c'est à dire sous une forme qui s'intègre facilement à tout système CORBA.

L'OGS fournit à tout objet CORBA la possibilité de créer un groupe, de s'y inscrire ou de s'en retirer, d'émettre des messages à destination d'un groupe et de recevoir les messages destinés au groupe auquel il appartient. En utilisant l'OGS, un objet peut, par exemple, implémenter des mécanismes de réplication.

On voit sur la figure 16 les différentes entités qui prennent part dans un échange de messages entre un client et un groupe de serveurs. Le client utilise le service d'un accesseur de groupe (*GroupAccessor)* pour envoyer des messages au groupe. L'accesseur de groupe communique avec les administrateurs de groupe (*GroupAd-*

ministrators) à travers le protocole interne de communication de groupe, qui est implémenté au-dessus d'IIOP. Les administrateurs de groupe délivrent les messages aux différents membres du groupe. A cette fin, les membres du groupe doivent implémenter l'interface *Groupable* et sont liés à travers elle à leur administrateur. Cette interface définit la méthode *deliver()* qui permet à l'administrateur d'appeler le membre dont il est responsable pour lui délivrer le message.

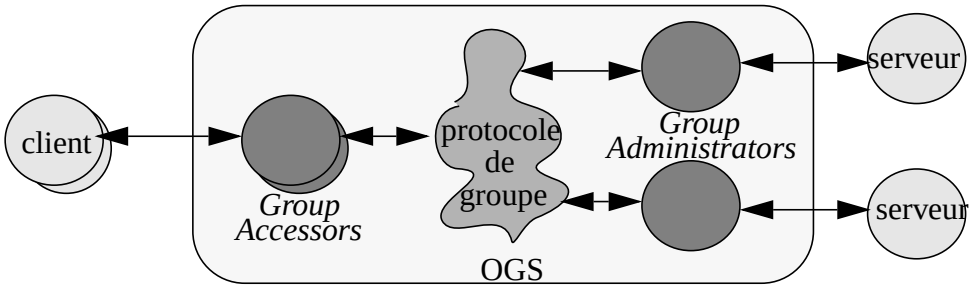

Figure 16 - Architecture du service OGS d'OPENDREAMS

Cette approche permet de bien séparer la gestion des protocoles de communication de groupe de l'application elle-même. Néanmoins, elle n'est pas transparente puisque les serveurs doivent implémenter l'interface *Groupable* afin de pouvoir utiliser le service. Par conséquent, l'application doit explicitement appeler le service de groupe pour son inscription ou pour envoyer un message.

Dans une récente évolution [Guerraoui *et al.* 1998], OPENDREAMS propose d'utiliser une forme d'interception des requêtes, afin d'implémenter la gestion du service de groupe et de mécanismes de tolérance aux fautes dans des entités différentes de l'application (cf. figure 17). Cette approche offre bien des similitudes avec l'approche par interception (cf. II.2.3) ainsi qu'avec le système GARF dont elle est issue (cf. II.2.6.2.1). En attendant que les intercepteurs CORBA soient spécifiés et disponibles (cf. I.2.2.3), le système utilise des techniques d'interception spécifiques aux ORB ORBIX [IONA 1997] et VISIBROKER [Visigenic 1997], à la manière de l'approche dite par interception (cf. II.2.3), c'est à dire en interceptant les messages qui transitent entre l'ORB et le système d'exploitation. Les messages émis du client vers le serveur sont déroutés vers le facteur (*mailer)*. Celui-ci utilise l'OGS afin d'émettre ces messages vers le groupe de répliques. Les messages reçus par le serveur sont déroutés vers l'encapsulateur (*encapsulator)*. Ce dernier traite cette requête selon le mécanisme de réplication sélectionné.

Cette dernière approche est bien plus transparente puisque *mailers* et *encapsulators* implémentent les mécanismes de tolérance aux fautes de façon séparée et indépendante de l'application. En revanche, l'approche perd en portabilité puisqu'elle utilise des mécanismes d'interception spécifiques à l'ORB. Ce dernier point devrait néanmoins, dans un avenir proche, disparaître puisque l'OMG prévoit la spécifica-

tion des intercepteurs. Enfin, l'utilisateur doit, s'il veut utiliser des mécanismes de recouvrement, implémenter des méthodes *SaveState* et *RestoreState* au sein des serveurs, comme avec la majorité des approches que nous présentons dans ce chapitre.

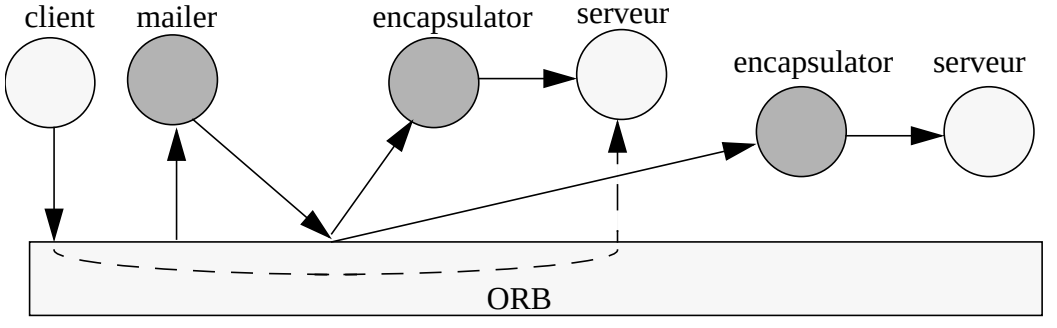

Figure 17 - Mécanisme d'interception dans OPENDREAMS

II.2.6 Approches langage

Enfin, certaines approches proposent d'implémenter les mécanismes de tolérance aux fautes au dessus des différentes couches de la plateforme, au niveau du langage. C'est le cas des approches à base d'héritage, qui exploitent les facilités de réutilisation fournies par les langages orientés-objet, ainsi que des approches réflexives, qui proposent d'utiliser la réfléxivité comme mécanisme fédérateur entre les objets applicatifs et les mécanismes non-fonctionnels. De ce fait, elles contraignent l'utilisateur à utiliser un langage d'implémentation particulier, orienté-objet ou réflexif. En revanche, puisqu'elles se placent à un haut niveau d'abstraction dans le système, elles sont, a priori, indépendantes des couches basses, système d'exploitation ou ORB. Cet avantage se transforme en inconvénient lorsqu'il s'agit d'obtenir des informations internes à ces couches basses.

II.2.6.1 Approches à base d'héritage

L'idée sous-jacente des systèmes de tolérance aux fautes à base d'héritage est d'utiliser cette technique pour ajouter et combiner diverses stratégies de tolérance aux fautes aux objets de l'application. Des classes, issues d'une hiérarchie complète, définissent des fonctions de base et des méthodes virtuelles que l'utilisateur devra, respectivement, utiliser et implémenter. Ces classes doivent donc être héritées par les classes de l'application. Ces dernières doivent correctement implémenter les méthodes virtuelles telles que *SaveState* et *RestoreState*, par exemple. Elles doivent aussi faire appel, au moment opportun, aux méthodes de leurs classes parentes, afin de lancer le processus de sauvegarde de l'état, par exemple.

Les systèmes ARJUNA [Shrivastava *et al.* 1991] et AVALON/C++ [Detlefs *et al.* 1988] sont deux exemples d'une telle approche. Ils utilisent l'héritage pour implémenter des actions atomiques. La notion d'action atomique permet d'accéder à un ensemble d'objets de manière contrôlée de telle façon que les actions effectuées sur

l'ensemble d'objets soient atomiques ; les modifications induites par les actions ne sont prises en compte que lorsque la dernière action aboutit ; si une des actions n'aboutit pas, l'ensemble des objets reprend son état initial, c'est-à-dire l'état qu'il avait avant que la première des actions ne débute.

Les actions atomiques permettent de tolérer les fautes physiques, omission de message ou défaillance par arrêt, ainsi que certaines situations logicielles bloquantes, par exemple, lors d'accès à des ressources partagées. Ces mécanismes reposent sur des techniques de sauvegarde sur support stable : avant qu'une action ne soit débutée, l'état des objets est sauvegardé sur un support non-volatile, si l'ensemble des actions ne peut être effectué, l'état des objets est restauré à partir de cette sauvegarde.

Le système ARJUNA définit une hiérarchie de classes qui permet la persistance des objets, l'accès exclusif à des ressources partagées et les actions atomiques. L'utilisateur qui désire fournir à une de ses classes une des propriétés énoncées ci-dessus doit alors la faire hériter de la classe correspondante, définir les méthodes *SaveState* et *RestoreState* et appeler correctement les fonctions de gestion des actions atomiques (activation/désactivation, préparation et validation de l'action, etc.).

On peut voir que cette approche est proche de celle à base de librairies. En effet, l'utilisateur doit implémenter lui-même certaines méthodes, en particulier celles concernant l'état de ses objets, et doit également faire appel aux fonctions relatives aux mécanismes proprement dits.

II.2.6.2 Approches réflexives

Une approche récente et prometteuse consiste à utiliser la réflexivité comme principe fédérateur entre l'application et les mécanismes non-fonctionnels, notamment pour la tolérance aux fautes. Plusieurs systèmes exploitent cette approche : le système MAUD [Agha *et al.* 1993] qui est basé sur un support d'exécution réflexif, le système GARF [Garbinato *et al.* 1995] qui, pour sa part, utilise une forme de réflexivité fournie par le langage SmallTalk et le système FRIENDS [Fabre et Pérennou 1998] qui utilise un langage réflexif, OpenC++ v1, pour ajouter des mécanismes non fonctionnels à des applications C++ standards.

II.2.6.2.1 MAUD et GARF

MAUD est basé sur le modèle à acteurs : une application distribuée est constituée d'un ensemble d'acteurs qui coopèrent par échange de messages. Le support d'exécution de MAUD est réflexif dans le sens où les messages échangés entre les acteurs applicatifs (les objets) peuvent être interceptés et traités par les acteurs systèmes (les métaobjets). Les liens entre les différents acteurs se font par connexion de ports, ces connexions étant dynamiques. Cela permet l'installation de mécanismes de tolérance aux fautes, autrement dit de métaobjets, de façon dynamique. De plus, les métaobjets peuvent être composés entre eux.

L'implémentation de ce système utilise le langage à acteurs HAL. Dans ce langage, les messages envoyés par un acteur sont transmis à une entité appelée *dispatcher*, responsable du routage des messages vers leur destination. Plusieurs variantes de cette entité permettent d'implémenter différentes versions de l'envoi d'un message : synchrone, asynchrone, etc. MAUD définit donc un *dispatcher* «réflexif». Celui-ci fait transiter les messages issus d'un acteur par le métaobjet qui lui est associé. Deux types de métaobjets peuvent être associés à un acteur, les *distributeurs* et les *collecteurs*. Un *distributeur* reçoit les messages d'un client et les retransmet aux *collecteurs* associés aux serveurs (cf. figure 18). Plusieurs mécanismes de réplication ont été implémentés grâce à cette approche, ils peuvent être composés entre eux en appliquant récursivement la réflexivité aux métaobjets eux-mêmes.

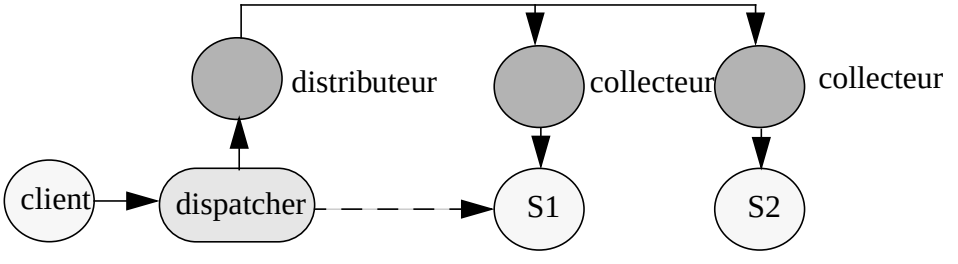

Figure 18 - Mécanisme d'interception dans MAUD

En ce qui concerne l'état des objets à des fins de clonage ou de réplication semi-active, l'utilisateur doit implémenter lui-même les méthodes nécessaires.

Cette approche est aisément configurable puisqu'il est possible de redéfinir à la fois les *dispatchers* et les métaobjets, il devrait donc être possible, dans l'absolu, d'utiliser un ORB du commerce, mais la combinaison et la coordination entre l'ORB et le support d'exécution risque d'être délicate.

Les mécanismes de tolérance aux fautes sont relativement transparents pour l'utilisateur si l'on exclut la nécessité d'implémentation des méthodes de gestion de l'état des objets. MAUD a été, à notre connaissance, le premier système à exploiter la réflexivité à des fins de tolérance aux fautes.

Pour sa part, le système GARF utilise un mécanisme d'interception inhérent au langage SmallTalk, mais il ressemble structurellement au système MAUD. Le métamodèle ainsi implémenté fut à la base de celui d'OPENDREAMS (cf. II.2.5), les propriétés de ces deux systèmes sont donc équivalentes. On notera toutefois que le système de communication de groupe utilisé dans GARF est ISIS [Birman *et al.* 1990].

II.2.6.2.2 FRIENDS

Le projet FRIENDS [Pérennou 1997][Fabre et Pérennou 1998] utilise Open-C++ v1 (cf. I.3.2.1) et son protocole à métaobjets simple pour ajouter de manière non-intrusive des mécanismes non-fonctionnels à des applications écrites en C++. Ces propriétés concernent la tolérance aux fautes (par réplication passive, semi-active ou active) et la sécurité par authentification de l'émetteur et le chiffrement des requêtes. L'approche suivie permet également d'isoler la gestion de la communication de

groupe dans un méta-niveau séparé. Chacune de ces propriétés est implémentée dans un métaobjet distinct, ces métaobjets pouvant être composés entre eux au gré de l'utilisateur (cf figure 19).

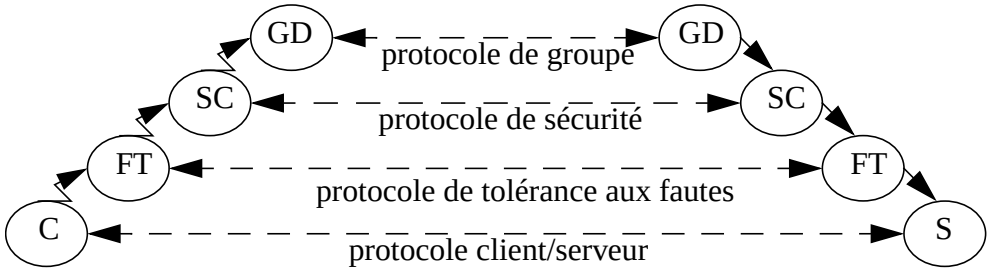

Figure 19 - Architecture de FRIENDS

Cette approche a démontré l'utilité des métaobjets pour satisfaire les besoins non fonctionnels d'un système. D'une part, les différents mécanismes peuvent être développés indépendamment de l'application et indépendamment les uns des autres. D'autre part, le fait de les concevoir à un niveau d'abstraction différent de celui de l'application (au méta-niveau) les rend suffisamment abstraits pour pouvoir être composés aisément, avec une intervention minimale de l'utilisateur. Ce système a, de plus, montré que la mise en place de mécanismes de tolérance aux fautes pouvait être combinée de façon élégante à l'utilisation d'un langage de programmation courant.

Néanmoins, quelques limites du système ont été identifiées, elles sont essentiellement liées au protocole à métaobjets utilisé et sont, pour la plupart, présentes dans les systèmes présentés précédemment :

- Objets et métaobjets sont liés statiquement. Il est difficile de configurer le système si seulement certaines instances d'une même classe doivent être attachées à un métaobjet. Il en va de même pour utiliser différents types de métaobjet avec différentes instances d'une même classe. Il est, de plus, impossible de changer le métaobjet d'un objet pendant l'exécution.
- La méta-information fournie par Open-C++ v1, bien que plus riche que dans les exemples précédents, reste incomplète et interdit l'usage de certaines fonctionnalités du langage comme, par exemple, l'héritage. L'application ne jouit donc pas pleinement de la transparence que pourrait offrir le système et de toutes les facilités du modèle à objet.
- Lors d'une invocation (cf. figure 7, page 21), Open-C++ v1 ne sait empaqueter et dépaqueter, sans intervention de l'utilisateur, que les variables de type simple. Les classes de l'application doivent donc se limiter à des attributs de type simple sans quoi le système ne peut déterminer leur état et donc ne peut pas les cloner.
- Un métaobjet ne peut accéder qu'à son propre niveau de base. Dans l'exemple de la figure 19, le métaobjet *SC* ne peut accéder à l'objet *S*. Par conséquent, dans certains cas, différentes propriétés doivent être composées au même méta-niveau et ne sont donc plus indépendantes les unes des autres.

FRIENDS utilise un protocole à métaobjets général, fourni par OpenC++ v1, pour adjoindre de façon statique des métaobjets à des instances de classes C++ standards. Cette approche bénéficie donc d'une relative transparence, aux conventions de programmation près. Le protocole à métaobjets d'OpenC++ v1 est utilisé comme lien fédérateur entre objets et métaobjets. Grâce à celui-ci, les métaobjets ont la possibilité d'obtenir l'état des objets (constitué d'attributs de type simple). Les métaobjets sont considérés comme des objets à part entière et peuvent donc être liés à leur propre métaobjet. La distribution de FRIENDS repose, non pas sur un ORB, mais sur des protocoles de communication de groupe fournis par le logiciel xAMp [Veríssimo et Marques 1990].

II.2.7 Evaluation des différentes approches

Dans cette section, nous cherchons à comparer les différentes approches selon les critères suivants : transparence, séparation, composabilité, visibilité, réutilisabilité et portabilité. Les points suivants définissent ces critères et les évaluent pour chacun des systèmes que nous avons présentés. Enfin, le tableau 1 résume ces résultats.

- **Transparence** : la transparence est la capacité d'une entité à être «invisible». Dans notre cas, elle cache à l'utilisateur les mécanismes non-fonctionnels du système.
  - Les approches à base de librairies, par intégration, par service et par héritage n'offrent pas un niveau de transparence satisfaisant. Avec celles-ci, l'utilisateur participe activement à l'implémentation des mécanismes de tolérance aux fautes.
  - Les systèmes DELTA4, ELECTRA et OPENDREAMS (lorsque celui-ci est couplé à l'interception) demandent uniquement à l'utilisateur d'intervenir pour la définition des fonctions de sauvegarde et de restauration de l'état.
  - FRIENDS est le seul système où, sous certaines conditions, l'état des objets est géré automatiquement.
- **Séparation et Composabilité** : la séparation correspond au degré d'indépendance existant entre l'implémentation des mécanismes de tolérance aux fautes et celle de l'application. Séparation et transparence sont des notions intimement liées. En effet, si l'approche n'est pas transparente, elle requiert une utilisation explicite des mécanismes dans l'application, ce qui nuit à l'indépendance et donc à la séparation entre ces entités. La composabilité correspond à la possibilité de combiner plusieurs mécanismes de nature différente. Evaluer la composabilité d'une approche n'a donc de sens que lorsque celle-ci permet la séparation. En effet, avec une approche qui ne permet pas la séparation, l'utilisateur implémente ses propres mécanismes au sein de l'application, leur composition est donc implicite.
  - Lorsque l'utilisateur utilise des librairies, les mécanismes de tolérance aux fautes sont fortement couplés à son application.
  - Avec les approches à base d'intégration, de service ou d'héritage, ce couplage est moins fort mais reste néanmoins peu satisfaisant.

  - En revanche, avec les approches système ou à interception, l'implémentation des mécanismes est bien séparée de celle de l'application. Il semble toutefois que la composition des mécanismes ne soit pas prise en compte dans ces différents systèmes.
  - Enfin, une approche réflexive, grâce à son protocole à métaobjets qui tient lieu de lien fédérateur transparent et souple entre mécanismes et application, permet à la fois transparence et composabilité des mécanismes.

- **Visibilité** : La visibilité concerne la façon avec laquelle les mécanismes sont visibles et configurables. Si plusieurs types de réplication sont disponibles, la visibilité permet à l'utilisateur de sélectionner celui qui est le plus adapté à son système ou à son application. Ce critère n'est pas incompatible avec la transparence. En effet, il est important que l'utilisation des mécanismes soit transparente au programmeur d'application (simplicité, efficacité, réutilisation du code applicatif), mais aussi que les mécanismes soient aisément accessibles du point de vue de leur configuration pour une application donnée.
  - Lorsqu'il utilise des librairies ou une approche à base d'héritage, l'utilisateur implémente lui-même ses propres mécanismes et, par conséquent, les contrôle parfaitement. Ces deux approches fournissent donc une visibilité à outrance. Cet aspect peut avoir des effets pervers sur le bon fonctionnement des mécanismes.
  - La visibilité est généralement bonne dans les autres cas. Mais, seul un mécanisme d'interception basé sur la réflexivité permet un bon compromis entre visibilité et transparence. Les mécanismes sont transparents pour l'utilisateur au niveau applicatif, mais sont visibles au méta-niveau et peuvent donc coopérer et être configurés aisément.
- **Réutilisabilité** : La réutilisabilité correspond à la possibilité donnée au concepteur du système de dériver de nouveaux mécanismes à partir des précédents et de les intégrer facilement au système.
  - Lorsque les mécanismes sont implantés statiquement dans le système, la librairie ou l'ORB (intégration), ils sont plus difficiles à réutiliser. C'est encore plus flagrant lorsque l'utilisateur participe à leur implémentation (mauvaise séparation).
  - Lorsqu'en revanche ils sont dynamiquement présents dans le système, comme c'est le cas pour les techniques à base d'interception, de service ou de réflexivité, leur réutilisation et modification est facilitée. L'approche à base d'héritage permet quant à elle une bonne réutilisation du code des mécanismes.
- **Portabilité** : l'approche est portable si elle ne repose pas sur un matériel, un système d'exploitation, un ORB, un support d'exécution ou un langage particulier ou spécifique. L'intérêt d'utiliser un système portable est évident : on pourra le réutiliser dans plusieurs environnements sans avoir à modifier ni l'approche, ni même l'implémentation des mécanismes ou de l'application.

- DELTA4 et ETERNAL figent les mécanismes dans une implémentation basée sur des systèmes d'exploitation et ORBs particuliers, ils sont donc difficilement portables.
- Les autres approches sont plus ouvertes à la portabilité car elles sont moins liées à une implémentation particulière.
- Un service tel que celui défini dans OPENDREAMS est hautement portable puisqu'il ne repose que sur les spécifications de la norme CORBA.

Approche	Transparence	Séparation	Visibilité	Réutilisabilité	Portabilité
Noyau/Système	+	+	+	-	-
Librairies	--	--	++	-	+
Interception	+	+	+	-	+
Intégration	--	-	+	+	-
Services	--	-	+	+	++
Services + Interception	+	+	+	+	+
Héritage	--	-	++	+	+
Réflexivité	++	++	+	++	+

Tableau 1 - Propriétés des différentes approches de la tolérance aux fautes dans les systèmes distribués

Symbole	Légende
--	pas du tout satisfaisant
-	peu satisfaisant
+	satisfaisant
++	très satisfaisant

II.3 Conclusion

Dans ce chapitre, nous avons présenté différentes approches pour la tolérance aux fautes dans les systèmes distribués à objets. Les différences majeures entre ces approches sont constituées par, d'un côté, la couche logicielle du système dans laquelle sont intégrés les mécanismes, et d'un autre côté, la technique d'obtention des informations relatives au comportement et à l'état de l'application. Nous avons ensuite comparé ces approches selon différents critères : transparence et séparation entre mécanismes et application, visibilité et configurabilité des mécanismes, réutilisabilité de l'architecture et des mécanismes lors du développement de nouveaux mécanismes et, enfin, portabilité du système sur une plate-forme différente.

Parmi ces différentes approches, et selon les critères que nous nous sommes fixés, l'approche la plus prometteuse est, sans aucun doute, l'approche réflexive. En effet, cette approche permet d'ajouter élégamment des mécanismes à une application dis-

tribuée, de façon transparente, portable et configurable. Cette approche offre également une très bonne séparation entre mécanismes et application, ou entre les mécanismes eux-mêmes.

FRIENDS est le seul système qui offre des mécanismes certes limités mais permettant dans leur principe de gérer automatiquement l'état des objets. Ce système utilise un protocole à métaobjets à l'exécution issu d'OpenC++ v1. Ses limitations viennent en grande partie de ce protocole à métaobjets et non de ses principes de base. Nous proposons dans cette thèse d'utiliser un protocole à métaobjets à la compilation qui, grâce à la richesse des informations qu'il fournit, permet de dépasser ces limites : la gestion de l'état des objets ne doit pas être limitée aux attributs de types simples, les métaobjets doivent pouvoir être dynamiques, la communication inter-objets doit se faire sur CORBA. L'usage de CORBA pour la communication entre objets et métaobjets permet, de plus, l'interopérabilité des langages : un objet écrit en C++ doir pouvoir être associé à un métaobjet développé en Java, par exemple. Cette interopérabilité permet une diversité qui est bénéfique pour la sûreté de fonctionnement.

Dans la suite de cette thèse, notre propos se concentre donc sur l'utilisation de la réflexivité à la compilation pour réaliser des protocoles à métaobjets spécifiques, qui fonctionnent à l'exécution. Bien que l'illustration de cette nouvelle technologie à base de compilateurs ouverts concerne ici les systèmes tolérants aux fautes, la méthode est générique et applicable à d'autres contextes.

Dans l'absolu, pour réaliser un système tolérant aux fautes dans une architecture CORBA, la meilleure solution consisterait à ne pas devoir modifier l'implémentation de l'application mais à utiliser des mécanismes d'interception et de capture de l'état intégrés à l'ORB. Malheureusement, ceci n'est pour l'instant possible qu'en développant sa propre plate-forme (système d'exploitation et ORB réflexifs), réduisant à néant la portabilité du système. En attendant une éventuelle standardisation de tels concepts et, puisque nous nous intéressons dans cette thèse à une plate-forme CORBA «boîte-noire», modifier l'implémentation des applications en utilisant la réflexivité à la compilation est une solution très attractive.

CHAPITRE III
CONCEPTION ET DÉFINITION D'UN PROTOCOLE À MÉTAOBJETS SPÉCIFIQUE

Dans le chapitre précédent, nous avons présenté différentes approches de la tolérance aux fautes dans des systèmes à objets distribués. Nous avons également motivé, en conclusion des précédents chapitres, l'utilisation de la réflexivité à la compilation pour créer un protocole à métaobjets à l'exécution qui soit vraiment adapté à des mécanismes de tolérance aux fautes pour des objets CORBA.

Dans ce chapitre, nous proposons de spécifier ce protocole à métaobjets. Pour cela nous étudions, tout d'abord, différents mécanismes de tolérance aux fautes afin de déterminer les informations qui leur sont nécessaires. Puis nous étudions les spécificités des objets CORBA du point de vue de ces mécanismes. Dans un second temps, nous examinons les différents moyens d'obtenir les informations nécessaires à ces mécanismes, et ce, en fonction du niveau de réflexivité du système et du langage utilisé. Enfin, nous donnons une définition du protocole à métaobjets dédié à la tolérance aux fautes des objets CORBA que nous réalisons au chapitre suivant.

III.1 Méta-informations et techniques de tolérance aux fautes

Dans cette section, nous étudions différents mécanismes de tolérance aux fautes afin d'identifier l'information réflexive nécessaire à leur implémentation pour des systèmes CORBA. Nous présentons, tout d'abord, brièvement ces mécanismes. Puis, par la suite, nous analysons l'information qui leur est nécessaire selon deux axes : comportement et état des objets. Enfin nous présentons les spécificités du modèle à objet CORBA qui influent sur la définition de notre protocole à métaobjets.

III.1.1 Techniques de tolérance aux fautes

Les techniques de tolérance aux fautes les plus utilisées sont basées sur la réplication de l'application ; la réplication utilise différentes techniques pour sauvegarder l'état de l'application, par exemple, des techniques de sauvegarde de points de reprise (*checkpointing)*.

III.1.1.1 La sauvegarde sur support stable

Avec cette technique, l'état de l'application est sauvegardé périodiquement sur un support stable, c'est-à-dire sur un support persistant qui conservera ces données en cas de défaillance du système, un disque dur RAID par exemple. Après une défaillance et lors de la restauration du système, l'application récupère son état à partir du support stable.

III.1.1.2 La réplication passive

Avec cette technique, l'application est répliquée, c'est à dire que plusieurs répliques de l'application sont exécutées sur différents nœuds du système. Une réplique est dite active, car elle traite les requêtes normalement puis envoie périodiquement une copie de son état à une ou plusieurs autres répliques. Ces dernières répliques sont dites passives, car elles ne traitent pas les requêtes et se contentent de recopier l'état qu'elles reçoivent de la réplique active. En cas de défaillance de la réplique active, une des répliques passives est élue pour devenir active, elle clone alors une nouvelle réplique passive et l'application peut ensuite reprendre normalement. Si c'est une réplique passive qui défaille, la réplique active clone simplement une nouvelle réplique passive sur un nœud adéquat du système.

III.1.1.3 La réplication active

Tout comme avec la réplication passive, l'application est répliquée en plusieurs répliques. En revanche, toutes les répliques traitent les requêtes. Une des réponses, parmi toutes celles qui proviennent des différentes répliques, doit alors être sélectionnée. Cette sélection s'effectue, soit à la source par une des répliques, soit à la destination par l'émetteur de la requête. La sélection peut éventuellement servir à la détection des défaillances. En effet si une des répliques fournit une réponse différente de celles des autres elle peut, sous certaines hypothèses, être considérée comme ayant défailli. Afin de pouvoir faire cette sélection, le nombre minimal de répliques est de trois, car il est impossible de déterminer la réponse exacte entre seulement deux réponses différentes. On peut noter aussi que cette forme de réplication nécessite des applications totalement déterministes, une même requête doit avoir une et une seule réponse possible pour que la sélection soit possible.

III.1.1.4 Conclusion

Les évènements et informations clés liés à ces différentes techniques de réplication peuvent aisément être identifiés (cf. tableau 2) : certains sont relatifs au comportement de l'application comme le fait de recevoir une requête et d'y répondre et d'autres touchent à l'état de l'objet afin de pouvoir le copier. Les paragraphes suivants détaillent ces informations selon cette classification.

	Etat des Objets	Comportement des objets
Sauvegarde sur support stable	✔	
Réplication passive	✔	✔
Réplication active	✔ (clonage)	✔

Tableau 2 - Informations nécessaires aux techniques de tolérance aux fautes

III.1.2 Informations liées au comportement des objets

Bien que la définition de points de reprise corresponde à la capture de l'état de l'application, le comportement de celle-ci, comme nous le verrons, influe grandement sur cette capture. C'est pourquoi nous nous intéressons ici aux différentes techniques à points de reprise, afin d'identifier les informations relatives au comportement des objets nécessaires à la reprise qui devront, en conséquence, être fournies (réifiées) par notre protocole à métaobjets.

III.1.2.1 Techniques de reprise

Alors qu'il est relativement aisé de prendre l'état d'une application non-répartie, il n'en va pas de même pour les applications distribuées. En effet, l'état d'une application distribuée dépend de l'état de plusieurs objets exécutés sur différentes machines et communiquant entre eux. Il est impossible de prendre l'état de tous ces objets au même instant pour produire un *état global cohérent*. Pour contourner ce problème, on doit utiliser différentes techniques pour établir, à partir d'une série d'états locaux, un état global cohérent du système [Jalote 1994]. On peut séparer ces techniques de reprise en deux approches. L'une est dite *asynchrone* car la prise des points de reprise individuels n'est pas coordonnée mais suffisamment d'information est fournie pour reconstruire un état global cohérent. La seconde approche, a contrario, coordonne la prise des points de reprise individuels pour former un état cohérent du système, elle est dite *synchrone*. Ces deux approches ont leurs avantages et leurs inconvénients.

L'approche asynchrone permet d'utiliser des points de reprise individuels de façon simple et peu coûteuse mais elle ne limite pas le nombre de pas en arrière que le système devra faire avant de trouver un état cohérent. Le système devra éventuellement conserver tous les points de reprise individuels effectués depuis son lance-

ment. Cette approche est donc plus coûteuse en stockage qu'en calcul lors du fonctionnement normal du système. En revanche, en cas de reprise, elle nécessite beaucoup de calculs afin d'établir l'état global cohérent du système.

L'approche synchrone, quant à elle, complique et augmente grandement le coût de la prise de points de reprise, car les différents objets doivent s'accorder avant d'acquérir leur point de reprise local. En revanche, le nombre de ces points de reprise est limité et on peut, une fois un point de reprise établi, assurer que le système pourra y revenir car ce point de reprise est cohérent avec l'état global du système. A l'inverse de l'approche asynchrone, l'approche synchrone est plus coûteuse en fonctionnement normal qu'en cas de reprise. En effet, lors de cette reprise, un état global cohérent du système est disponible et n'a donc pas besoin d'être recherché.

Au delà des différences entre ces deux approches, on peut définir un ensemble d'informations à partir desquelles elles pourront fonctionner. Si l'on considère qu'un état global cohérent est formé d'un ensemble d'états locaux pour lesquels tout message envoyé a été reçu, cela implique :

- qu'il n'existe pas d'objet dans un état où le message envoyé n'a pas été reçu, car le **message** serait **perdu**, son récepteur ne le recevrait jamais.
- qu'il n'existe pas d'objet qui soit dans un état où il a reçu un message alors que son émetteur est dans un état antérieur à l'émission (comme si le message n'avait pas été envoyé), car le **message** serait **orphelin**, son émetteur pourrait le réémettre et ainsi perturber le récepteur.

Pour des objets, l'envoi ou la réception de message correspondent tous deux à l'invocation d'une méthode, respectivement l'objet invoque une méthode sur un objet distant ou une méthode de l'objet est invoquée par un objet distant. C'est pourquoi il est nécessaire de réifier ces deux types d'information: invocation entrante **et** invocation sortante.

On notera toutefois que certains algorithmes de reprise, comme celui de Koo et Toueg [Koo et Toueg 1987], s'assurent de la non-occurrence de perte de message en faisant l'hypothèse d'un canal de communication fiable. Le protocole de reprise synchrone qu'ils proposent est donc simplifié et rendu plus efficace par cette hypothèse. Dans ce cas, puisque tout message envoyé est reçu, seule la réification des invocations entrantes est nécessaire.

III.1.2.2 Clonage

Les différentes techniques de réplication ont toutes besoin de pouvoir cloner des objets ; que ce soit à l'initialisation du système pour créer les répliques initiales ou lors de la reconfiguration de celui-ci après une défaillance pour ramener le nombre de répliques à son niveau précédent. Cloner un objet implique deux choses : d'une part, le système doit connaître l'état de l'objet à cloner (nous discuterons ce point dans les paragraphes suivants) et d'autre part, il doit savoir créer un nouvel objet. C'est ce dernier point qui nous intéresse ici puisqu'il touche tout particulièrement au comportement des objets.

En effet, un objet est créé par une méthode particulière appelée constructeur. Ce constructeur est responsable de l'initialisation des différents attributs de l'objet. Un métaobjet qui désire créer un nouvel objet doit donc pouvoir appeler ce constructeur.

De plus, le processus de création des répliques est utilisé dès la création de la première réplique. Supposons, en effet, que la stratégie de réplication précise qu'un certain objet doit être répliqué en trois exemplaires ; sur quel critère décide-t'on de démarrer la réplication? Un évènement doit initier le processus, cet évènement sera pour nous la création de la première réplique. Notre protocole à métaobjets devra donc réifier la création des objets, autrement dit l'appel au constructeur.

Parallèlement à la création des objets, et afin de conserver un système cohérent ainsi que pour ne pas saturer inutilement les ressources du système, les répliques doivent pouvoir être détruites. Pour ce faire le protocole à métaobjets devra réifier l'appel au destructeur des objets et permettre au métaobjet d'accéder à celui-ci. Ainsi la destruction de la réplique initiale pourra entraîner la destruction des autres répliques, ceci reste toutefois à la discrétion de leurs métaobjets respectifs et du protocole qu'ils implémentent.

III.1.2.3 Conclusion

Comme on le voit sur le tableau 3, on distingue quatre types d'information comportementale. La réification des invocations entrantes est nécessaire aux techniques de reprise asynchrone et synchrone. Ces invocations entrantes correspondent en effet à la réception d'un message. La réification des invocations sortantes, en revanche, n'est pas toujours nécessaire. En effet, certaines techniques de reprise synchrone peuvent se passer de la réification de l'émission d'un message. Enfin, l'accès aux constructeurs et destructeurs est non seulement nécessaire aux différentes techniques de reprise mais aussi au clonage des objets.

	Invocation entrante	Invocation sortante	Constructeur	Destructeur
Reprise asynchrone et clonage	✔	✔	✔	✔
Reprise synchrone et clonage	✔	✔/-[a]	✔	✔

Tableau 3 - Informations comportementales nécessaires aux techniques de reprise

a. Certains algorithmes de reprise synchrone peuvent se passer de cette information (cf. III.1.2.1)

III.1.3 Etat

Comme nous l'avons vu précédemment, savoir obtenir l'état local d'un objet distribué est nécessaire pour pouvoir construire l'état global cohérent du système. En effet, ce dernier est construit à partir d'un ensemble des états des objets du système. Nous devons donc définir ce qu'est l'état d'un objet distribué et comment l'obtenir.

III.1.3.1 Reprise non orientée-objet

Dans le cas simple d'une application non-objet, non-répartie, un état pourrait être formé des valeurs des variables du processus, de son environnement, de l'information de contrôle des tâches, de la valeur des registres du processeur, etc. De nombreux travaux ont été menés dans ce domaine, à partir d'approches systèmes, de librairies ou encore de compilateurs, elles sont comparées dans [Plank 1997].

Les approches systèmes, tel Delta-4 cf. II.2.1, proposent d'intégrer au système d'exploitation des mécanismes qui permettent de connaître les pages mémoires, registres et variables d'environnement utilisées par une application. Cette approche a le mérite d'être efficace, l'obtention des points de reprise étant effectuée à très bas niveau. En revanche, les données fournies par le système sont difficilement analysables et donc difficilement portables d'un système à un autre, même équivalent.

Les librairies fournissent des fonctions de sauvegarde de points de reprise génériques au sein de librairies auxquelles l'application doit être liée. Cette approche est en général moins transparente que la précédente puisque le programmeur d'application doit explicitement appeler ces fonctions. De plus, se plaçant au dessus du système d'exploitation, ce qui est le cas pour de nombreuses autres approches (cf II.2), la librairie ne peut pas sauvegarder l'état de celui-ci, cela pose problème lorsqu'une partie de l'état de l'application est contenue dans des variables internes du système. En effet, il arrive souvent qu'une application utilise des variables dites **locales** au sens où leur valeur n'a de sens que sur un système donné, à un instant donné. Un bon exemple de telles variables locales sont les fichiers Unix qui sont représentés dans les applications par un nombre entier ; ce nombre est généré par le système à l'ouverture d'un fichier et correspond à un index dans un tableau du système. Copier cet entier sur un autre site du système réparti ou même sur le même site à un autre instant n'a aucun sens. Lorsqu'il utilise de telles variables, le programmeur d'application doit donc spécifier à la librairie un moyen supplémentaire de sauver l'état de la variable en question ; dans le cas des fichiers il peut suffire de copier le nom et l'index des fichiers ouverts [Long *et al.* 1991]. Ces inconvénients sont à contre-balancer avec la souplesse qu'apporte l'approche; en effet, le programmeur peut plus facilement adapter la stratégie utilisée à son application.

Les approches à base de compilateurs se placent à un niveau d'abstraction encore supérieur puisqu'elles agissent directement au niveau du langage de programmation utilisé par l'application. Le compilateur analyse le code source de l'application et le réécrit de façon à ce qu'il incorpore les routines de sauvegarde et de restauration de l'état. Cette analyse permet d'adapter de façon très fine la stratégie à l'application

source. PORCH [Strumpen et Ramkumar 1998], par exemple, fournit un tel compilateur pour langage C qui produit des points de reprise dans un format indépendant du système de telle sorte que ceux-ci puissent être utilisés sur différentes architectures.

III.1.3.2 Etat interne des objets

Le fait d'utiliser un modèle à objets nous permet de nous abstraire de certaines des informations «système» puisque l'objet encapsule les informations dont il a besoin. Ceci est valable bien entendu uniquement dans un environnement purement objet où chaque entité du système est un objet. Grâce à cette hypothèse on peut dire que l'état d'un objet est constitué des valeurs de tous ses attributs. Et que l'état du système est constitué d'une collection d'états des objets du système formant un état global cohérent.

Nous pensons que l'hypothèse «tout objet» est ou sera plausible dans un futur proche car l'on voit se développer de plus en plus de systèmes d'exploitation orientés-objet : GUIDE [Balter *et al.* 1990] fut un des premiers systèmes à introduire la notion d'objet comme entité propre du système, au même titre qu'UNIX est basé sur la notion de processus. Plus tard ont été développés CHOICES [Campbell *et al.* 1993] ou plus récemment APERTOS [Yokote 1992], ce dernier utilisant un modèle à objets réflexif. Le développement massif de CORBA et de ses différents services, comme FRIGATE [Kim et Popek 1997] un système de fichiers sur CORBA, nous porte à croire qu'à terme tout service sera implémenté sous la forme d'objet.

Pouvoir fournir l'état d'un objet est donc une brique de base de tout algorithme de reprise, notre protocole à métaobjets doit donc être en mesure de fournir cet état. Grâce à l'encapsulation, nous avons vu que l'état d'un objet peut être vu comme l'ensemble des valeurs de ses attributs. Par la suite, nous appellerons cet ensemble de valeurs l'**état complet** de l'objet puisqu'il contient la valeur de tous les attributs de celui-ci.

Dans certains cas, dans un souci d'efficacité, il peut être souhaitable de réduire la quantité de données contenues dans un point de reprise. Dans cette optique, plusieurs travaux sur la notion de point de reprise incrémental ont été menés ; on peut citer notamment [Netzer et Weaver 1994] où seules les valeurs ayant été modifiées sont inclues dans les points de reprise. Leurs travaux portent sur le déboguage d'application : lors de celui-ci, il est parfois nécessaire de ré-exécuter plusieurs fois la même portion de code. A cette fin, les techniques de reprise sont utiles pour ré-exécuter l'application à partir d'un point précis. Elles évitent d'avoir à relancer l'application depuis son début, ce qui n'est pas toujours possible. En revanche, dans un tel contexte, on doit conserver un nombre important de points de reprise. On ne peut, en effet, pas déterminer à l'avance quel sera le point à partir duquel il faudra recommencer l'application. Une récriture automatique de l'application permet de faire en sorte que chaque variable modifiée soit marquée comme telle. Ainsi la fonction de production des points de reprise peut se contenter de ne prendre en

compte que les variables qui ont été effectivement modifiées. Cette technique réduit de manière significative la taille des points de reprise mais, en revanche, augmente de façon importante le temps d'exécution de l'application.

Nous proposons d'appliquer la même méthode aux objets, c'est à dire de fournir des **états partiels** qui incluent uniquement la valeur des attributs ayant été modifiés depuis la précédente sauvegarde de l'état. Il nous semble que dans bien des cas, peu d'attributs des objets sont modifiés entre deux appels de méthodes ou entre deux sauvegardes d'état. C'est pourquoi il peut être intéressant de fournir des états partiels aux applications pour lesquelles le temps de transfert ou de sauvegarde est bien plus critique que le temps d'exécution. Bien entendu nous n'imposons pas cette technique et elle devra être sélectionnée au cas par cas par l'utilisateur en fonction d'un compromis entre le coût des communications et des performances de l'application.

III.1.3.3 Etat externe des objets

Bien des protocoles de recouvrement arrière sont basés sur l'utilisation de points de reprise et de journaux pour restaurer une application à la suite d'une défaillance. Un point de reprise contient généralement une copie de l'état de l'application et suffisamment d'information pour reprendre l'application au moment où l'état a été pris. Le journal quant à lui contient de l'information à propos des évènements non-déterministes qui sont survenus avant la défaillance, par exemple les interactions avec l'utilisateur, les signaux UNIX, ou encore l'information relative aux variables dites locales de l'application, ordonnancement et synchronisation des tâches, fichiers ouverts, etc.

Plusieurs approches se détachent de ces différents protocoles. La plus simple consiste à sauvegarder l'état après chaque occurrence d'un évènement non-déterministe [Elnozahy 1993], ce qui peut être très coûteux. Une autre approche consiste à rejouer ces séquences d'évènements asynchrones de manière synchrone comme dans Delta-4 [Powell 1991]; cette approche requiert des modifications du système d'exploitation et peut ne pas être appropriée ou efficace pour certaines applications. Plus récemment, l'utilisation de librairies de tâches spécifiques a été proposée [Slye et Elnozahy 1996] ainsi qu'une instrumentation de l'application au niveau du code machine afin de pouvoir rejouer les évènements asynchrones; il est évident que cette technique est hautement intrusive et ne peut pas être utilisée lorsque l'on se place au dessus d'une abstraction de haut niveau comme un ORB.

Il semblerait que seule une gestion de ces évènements à un haut-niveau d'abstraction puisse fournir une solution qui serait à la fois transparente et souple. Dans Eternal par exemple, l'idée [Narasimhan *et al.* 1999] consiste à placer un ordonnanceur dans chaque domaine de réplication multi-tâche, cet ordonnanceur distribuant les requêtes aux différentes tâches de l'application de manière déterministe. Afin de pouvoir implémenter une telle approche, l'ordonnanceur doit avoir connaissance de toutes les informations concernant les tâches: création, activation, synchronisation, etc., ainsi que celles concernant les requêtes (invocations et réponses). En

attendant que l'ORB ne réifie de telles informations, grâce aux *Portable Interceptors* [OMG 1998e] en ce qui concerne les requêtes, ces travaux s'appuient sur une interception de messages ainsi qu'une librairie de gestion des tâches personnalisées.

III.1.3.4 Conclusion

Nous avons vu dans cette section que l'obtention de l'état des objets était une fonction nécessaire de tout système tolérant les fautes, et que celle-ci n'est pas aisée. L'état des objets est composé de deux parties : l'état interne qui est contenu dans l'objet lui-même et qui est constitué de l'ensemble des attributs, et l'état externe qui repose dans des couches sous-jacentes comme des librairies ou le système d'exploitation.

Obtenir l'état interne des objets n'est pas chose évidente mais est possible. Plusieurs techniques permettent même d'optimiser la taille de celui-ci en ne fournissant que l'état partiel de l'objet : l'ensemble des attributs qui ont été précédemment modifiés. C'est l'état externe qui pose le plus de problèmes : non seulement il est difficile à obtenir car le système d'exploitation ne propose pas un accès à ses variables internes mais il est d'autant plus difficile à identifier. En effet, cet état externe dépend en grande partie du système d'exploitation, de ses mécanismes et services internes.

Nous présentons dans le tableau 4, pour quels mécanismes sont nécessaires ces différentes informations concernant l'état des objets. On notera en particulier que même si la réplication active n'utilise pas directement l'état des objets, elle nécessite parfois le clonage de ces derniers. Le clonage quant à lui, nécessite l'état complet de l'objet.

	Etat Interne		Etat Externe
	complet	partiel	
Réplication active (et clonage)	✔		✔
Réplication passive (et clonage)	✔	✔	✔

Tableau 4 - Nécessité de l'état pour les techniques de réplication

III.1.4 Spécificités des objets CORBA

Notre protocole à métaobjets doit s'appliquer aux objets CORBA. Le modèle à objet de CORBA comporte certaines spécificités qui sont donc à prendre en compte dans la conception de ce protocole à métaobjets. Par exemple, le fait que chaque client effectue ses invocations au serveur à travers un mandataire (*stub*), ou encore que l'ORB fournisse un système de référence des objets, ont des incidences sur le protocole.

III.1.4.1 Communication entre client et serveur CORBA

Les objets CORBA sont définis d'une part par leur interface IDL et d'autre part par leur implémentation dans un langage donné (cf figure 20). Si le langage d'implémentation est un langage orienté-objet, comme c'est souvent le cas, une classe implémente l'interface IDL et possède elle-même une interface dans son langage natif. L'interface native est un sur-ensemble de l'interface IDL : elle contient aussi bien les méthodes de cette dernière que d'autres méthodes à usage plus restreint. Ces méthodes à usage restreint ne sont pas accessibles à travers l'ORB, elles le sont seulement dans le même espace d'adressage que l'objet lui-même, et ce uniquement si elles sont publiques. Nous avons choisi d'utiliser l'objet CORBA comme abstraction de base (comme unité de réplication), et c'est pourquoi nous considérons que d'un point de vue externe à l'objet, seule l'interface IDL est intéressante. Seules les méthodes appartenant à l'interface IDL devront être réifiées, on considérera que les invocations des autres méthodes sont internes à l'objet et donc correspondent à une structuration propre à l'implémentation de l'objet.

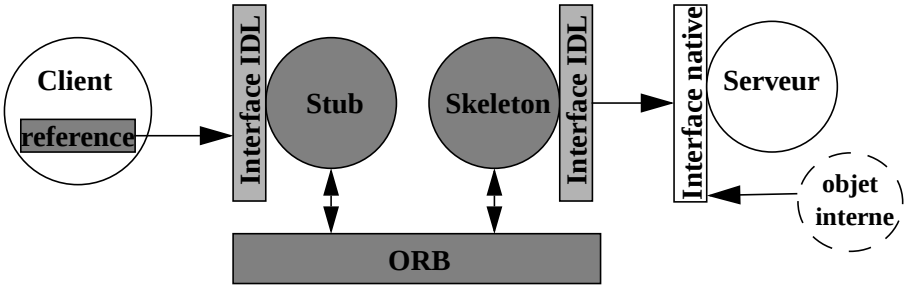

Figure 20 - Interfaces, Stubs, Skeletons et Références

On peut voir figure 20 qu'un client accède à un serveur à travers un *stub*, qui joue le rôle de mandataire, c'est à dire qu'il est le représentant du serveur dans l'espace d'adressage du client. Le *stub* effectue l'empaquetage et le dépaquetage des arguments ainsi que les appels à l'ORB. Dans les protocoles de tolérance aux fautes courants [Fabre et Pérennou 1998], le client joue un rôle non-négligeable, il peut par exemple devoir réémettre certaines requêtes, ou encore effectuer un vote sur les différentes réponses reçues suite à une requête. De plus, comme nous l'avons vu précédemment, la réification des invocations sortantes est nécessaire à certains mécanismes de réplication. Ces invocations sortantes correspondent, dans CORBA, à l'invocation d'une méthode d'un stub. C'est pourquoi notre protocole à métaobjets, comme nous le verrons, intègre des méta-stubs à qui les stubs réifient création, destruction et invocation.

D'autre part, le client accède au serveur à travers une référence sur celui-ci. Cette référence désigne pour l'ORB un objet unique. Dans notre contexte, la référence désigne un service qui peut éventuellement être implémenté par plusieurs objets répliqués. Pour que la défaillance de l'objet référencé soit tolérée par le système de manière transparente, la gestion de la référence par le *stub* doit donc masquer cette

défaillance. En plus des informations que nous avons identifiées ci-dessus, le méta-stub doit donc être capable de contrôler la référence gérée par le stub dont il est responsable.

III.1.4.2 Modes de concurrence

Les objets CORBA implémentent des services qui peuvent être accessibles à plusieurs clients. Par conséquent, plusieurs requêtes peuvent lui parvenir de manière concurrente. Dans l'architecture CORBA, c'est l'adaptateur d'objet (*Object Adapter)* qui a la responsabilité de l'activation des méthodes pour chaque interface, c'est donc lui qui sérialise ou non les invocations. La norme CORBA 2.0 ne spécifie que deux modes d'activation distincts: *SINGLE_THREAD_MODEL* et *ORB_CTRL_MODEL*. Le premier assure que chaque adaptateur n'a qu'une tâche de contrôle et donc doit sérialiser les requêtes parvenant à l'objet qui implémente l'interface qu'a en charge l'adaptateur. Le second, par contre, précise que l'ORB gérera le multi-tâche. Cette spécification est reconnue comme très floue [Schmidt et Vinoski 1997], elle devrait être révisée dans les prochaines versions de la norme. Dans l'état actuel des choses, il est très difficile de contrôler la politique de concurrence si le mode multi-tâche est sélectionné, puisque chaque ORB implémente une politique différente. De plus, il est possible à un objet d'implémenter plusieurs interfaces IDL et donc d'être contrôlé par autant d'adaptateurs différents ; ceux-ci peuvent donc invoquer de manière concurrente des méthodes de l'objet même si chacun est en mode *SINGLE_THREAD_MODEL*.

Comme nous l'avons vu en section III.1.3.3, les tâches affectées à un objet font partie de son état externe puisque cette information est conservée dans les couches inférieures du système (système d'exploitation ou ORB). Cette information doit donc être prise en compte dans les mécanismes de tolérance aux fautes lorsque c'est possible. Lorsqu'elle n'est pas disponible, il faut assurer que l'objet est mono-tâche.

III.1.4.3 Méthodes *oneway*

Une autre particularité du modèle à objet de CORBA est le fait que certaines méthodes peuvent être définies comme étant *oneway*, c'est à dire qu'elles ne génèrent aucune réponse à une invocation. Ces méthodes n'ont ni type de retour ni paramètre de sortie (*out)* ou d'entrée/sortie (*inout)*, on peut dire qu'elles sont asynchrones ou encore non-bloquantes.

Le client qui invoque une méthode oneway ne reçoit ni réponse, ni confirmation ; il n'y a pas moyen de s'assurer de la délivrance d'une telle requête. D'un point de vue de la sûreté de fonctionnement, ne pas avoir confirmation de la délivrance d'une requête pose problème.

Nous proposons que de tels appels soient réifiés de telle sorte que le métaobjet puisse s'assurer de la livraison du message. Ceci se fera tout en conservant la sémantique asynchrone de l'invocation.

III.1.4.4 Conclusion

L'étude des spécificités des objets CORBA nous a amené à identifier un point important du protocole à métaobjets : la nécessité de pouvoir contrôler les stubs au sein d'un métaobjet particulier appelé méta-stub. En effet, les stubs sont responsables des invocations sortantes des objets CORBA. Ils sont également responsables de la gestion des références sur les objets CORBA. Outre la création et la destruction des stubs, les méta-stubs doivent donc être capables de contrôler ces deux informations : invocation sortante et références.

Un deuxième point à ne pas négliger concerne la concurrence des objets applicatifs. En effet, CORBA permet d'utiliser des objets multi-tâches. Malheureusement, l'information de contrôle de ces tâches n'est pas disponible, nous devrons donc nous contenter d'objets mono-tâche. Enfin, les méthodes oneway (ou asynchrones), doivent être réifiées comme les méthodes standard, en conservant leur sémantique particulière.

III.2 Obtention de la métainformation

Maintenant que nous avons défini les informations qui nous sont nécessaires pour implémenter des protocoles de tolérance aux fautes au sein de métaobjets, nous allons étudier de quelle façon ces informations peuvent être obtenues. En effet, selon le système ou le langage utilisé, ces informations peuvent être plus ou moins aisément accessibles ; par exemple, un support d'exécution de langage réflexif fournira naturellement toutes ces informations alors qu'un support d'exécution «boite-noire» ne permettra pas d'y accéder.

Le protocole à métaobjets que nous définissons se veut totalement indépendant de la façon dont il sera implémenté, c'est pourquoi nous examinons dans cette section les différentes solutions qui peuvent ou pourront exister dans un futur proche comme par exemple avec de prochaines versions de la norme CORBA.

III.2.1 Informations relatives au comportement des objets

Nous avons identifié au paragraphe III.1.2 plusieurs types d'informations relatives au comportement des objets : celles liées à la création ou la destruction des objets, celles concernant l'invocation de méthode entrante et enfin celles relatives à l'invocation de méthode sortantes, i.e. sur des objets externes. Etudions maintenant les différentes possibilités que nous avons d'obtenir ces informations en fonction du type de support d'exécution utilisé : réflexif, non-réflexif ou partiellement réflexif.

III.2.1.1 C++ : un support d'exécution non-réflexif

Le langage C++ n'offre aucune réflexivité, y compris dans ses versions les plus récentes comme celle de la norme ANSI [ISO 1998]. Pour pallier ce manque de réflexivité, il est donc nécessaire d'utiliser des extensions du langage comme le fait

Open C++ version 1, ou encore utiliser un compilateur ouvert comme Open C++ version 2 (cf. I.3.2.2). Le premier fournit une réflexivité à l'exécution alors que le dernier une réflexivité à la compilation. L'utilisation de la réflexivité à l'exécution pour la tolérance aux fautes a déjà été traitée dans [Pérennou 1997], et discutée en section I.3.2.1; nous discuterons donc ici de la réflexivité à la compilation.

Comme nous l'avons décrit en I.3.2.2, l'utilisation d'un compilateur ouvert tel qu'Open C++ v2, permet d'analyser et de modifier en conséquence le code source d'une classe. Le but principal de ces modifications est de permettre la réification des informations comportementales des objets au métaobjet et de permettre l'intercession de ce dernier vers l'objet lui-même. Pour ce faire, l'utilisation de *wrappers*, ou encapsulateurs, est communément admise comme étant des plus pratiques : le code original de la classe est remplacé par un code qui effectue la réification, du code supplémentaire permet l'accès au code original pour implémenter l'intercession. Bien des techniques d'encapsulation existent ; pour le langage Smalltalk, sept sont décrites dans [Brant *et al.* 1998].

En ce qui concerne C++, le choix est plus limité : le code C++ est compilé statiquement alors que celui de Smalltalk peut l'être dynamiquement. En outre, Smalltalk offre des fonctionnalités de sélection de méthode dynamique que ne propose pas C++. De plus, les *wrappers* décrits dans [Brant *et al.* 1998] sont utilisés pour ajouter des appels à des méthodes *before/after* et non pour encapsuler totalement une méthode. Dans le cas de méthodes *before/after* on peut insérer directement les appels à ces dernières dans le code de la méthode originale. Cela n'est pas envisageable pour notre protocole puisque le métaobjet doit non seulement pouvoir contrôler l'objet avant et après l'invocation de la méthode, mais il doit aussi contrôler l'invocation elle-même, quitte à ne pas réaliser d'invocation du tout. Parmi les sept techniques proposées, seules deux sont applicables à notre problème : l'encapsulation de classe et l'encapsulation de méthodes. Chacune de ces deux techniques possède ses propres avantages et inconvénients, nous les discuterons dans les paragraphes qui suivent.

III.2.1.1.1 Encapsulation de classe

La méthode la plus simple pour changer le comportement d'une classe est sans aucun doute de substituer cette classe par une autre, différente. La nouvelle classe utilise le nom de la précédente qui, pour sa part, est renommée. Elle implémente la même interface que la classe originale, ainsi son utilisation est alors implicite et totalement transparente. Afin de pouvoir accéder au comportement original de la classe encapsulée, elle peut au choix en hériter ou utiliser la délégation. Si elle utilise l'héritage, elle peut appeler les méthodes originales grâce à la notation *ClasseDeBase::Méthode()*. Si, au contraire, elle utilise la délégation, elle peut appeler la méthode de son délégué grâce à un appel de la forme *Delegue.Methode()*. Le tableau 5 donne un exemple de chacune des deux approches.

Classe originale	**Classe originale encapsulée**

`class Original {`
`void foo() {`
`  ...`
`  }`
`};`
	`class WrappedOriginal {`
	`void foo() {`
	`  ...`
	`  }`
	`};`

Wrapper par délégation	

`class Originale {`
`  WrappedOriginale Delegue;`
`  void foo() {`
`  // Réification de l'invocation`
`  }`
`  void call_foo() {`
`  Delegue.foo();`
`  }`
`};`	

Wrapper par héritage	

`class Originale : private WrappedOriginal {`
`  void foo() {`
`  // Réification de l'invocation`
`  }`
`  void call_foo() {`
`  Originale::foo();`
`  }`
`};`	

Tableau 5 - Exemple d'encapsulation de classe

Le principal avantage de ces techniques est de ne nécessiter qu'une modification mineure de la classe originale : son renommage. Le renommage de la classe implique en C++ le renommage des constructeurs et du destructeur. Néanmoins le code original de la classe reste quasiment inchangé et le code qui implémente l'encapsulation est complètement séparé puisqu'implémenté dans une classe différente.

Bien qu'avantageuses à de nombreux points de vue, ces techniques comportent, toutes deux, des inconvénients rédhibitoires.

Le premier d'entre eux, pour la technique à base de délégation, est posé par l'auto désignation : si la classe originale utilise le mot clé *this* pour désigner l'instance courante, la classe qui l'encapsule ne sera pas appelée.

De plus, si l'application utilise le polymorphisme, l'approche par délégation ne peut pas être utilisée. En effet, les types de la classe qui encapsule et de la classe encapsulée ne sont pas compatibles. Notons qu'avec l'approche par héritage, le polymor-

phisme n'est que partiellement supporté : la classe encapsulée ne doit pas hériter de façon privée, sans quoi la classe qui encapsule ne peut accéder à la hiérarchie d'héritage et donc son type n'est pas compatible avec la classe mère.

Le dernier défaut est commun aux deux techniques, il apparaît lors de l'utilisation de méthodes ou d'attributs privés ou protégés. En effet, la classe qui encapsule ne peut y accéder (ou seulement aux protégés si l'approche utilisée est l'héritage) et donc ne peut implémenter l'encapsulation correctement.

On le voit, même si l'encapsulation de classe est séduisante au premier abord, son utilisation ne peut se faire de façon transparente pour l'utilisateur. En effet, ses limitations sont importantes et obligent l'utilisateur à observer de nombreuses et contraignantes conventions de programmation permettant de rester compatible avec l'approche.

III.2.1.1.2 Encapsulation de méthode

La deuxième technique d'encapsulation possible pour réifier le comportement des objets consiste à modifier directement les méthodes de la classe originale. En effet, il suffit de renommer les méthodes originales et de les remplacer par de nouvelles méthodes ayant la même signature (retour, nom, arguments) pour que l'encapsulation soit totale et transparente. Bien évidemment constructeurs et destructeur doivent être traités de la même manière.

Bien qu'en apparence plus simple, cette technique requiert de nombreuses modifications du code original. D'autant plus que nous ne souhaitons pas réifier les appels internes de l'objet sur lui-même, comme identifié en III.1.4.1, il faut donc traduire également ces appels pour qu'ils invoquent les méthodes renommées et non celles qui les encapsulent (cf. tableau 6).

Malgré de légers inconvénients, cette technique reste très intéressante. En effet, le fait que la classe utilisée soit la classe originale lève les ambiguïtés posées par la technique précédente. L'objet peut s'auto-désigner sans problème et peut accéder à ses méthodes privées. De plus, la non-utilisation de l'héritage améliore la testabilité du code résultant, et ce malgré les nombreuses modifications apportées à la classe originale.

Classe originale	Classe avec encapsulation des méthodes

`class Originale {`
`  void foo() {`
`  ...`
`  }`
`  int bar(float a) {`
`  ...`
`  foo();`
`  ...`
`  }`
`};`
	`class Originale {`
	`  void foo() {`
	`  // Réification de l'appel`
	`  }`
	`  int bar(float a) {`
	`  // Réification de l'appel`
	`  }`
	`  void foo_ori() {`
	`  ...`
	`  }`
	`  int bar_ori(float a) {`
	`  ...`
	`  foo_ori();`
	`  ...`
	`  }`
	`};`

Tableau 6 - Exemple d'encapsulation de méthodes

III.2.1.1.3 Le langage C++ et l'information comportementale : conclusion

Pour rendre réflexives des applications développées en C++, langage qui ne l'est pas, la seule solution consiste à utiliser un compilateur ouvert tel que celui d'OpenC++ v2. Un tel outil permet d'analyser le code source de l'application et de lui appliquer des transformations en fonction de cette analyse.

Dans cette section, nous avons présenté trois techniques d'encapsulation différentes qui sont applicables au langage C++ en utilisant un compilateur ouvert : encapsulation de classe par héritage ou par délégation et encapsulation de méthodes. Chacune de ces techniques ont leurs avantages et inconvénients qui sont résumés dans le tableau 7.

	Séparation	Auto-représentation	Polymorphisme	Accès aux attributs
Encapsulation de classe par héritage	✔	✔	✔	
Encapsulation de classe par délégation	✔			
Encapsulation de méthode		✔	✔	✔

Tableau 7 - Comparaison des différentes techniques d'encapsulation

Les techniques par encapsulation de classe nécessitent moins de modifications de la classe originale et séparent mieux l'entité encapsulatrice de la classe originale que la technique à encapsulation de méthode. En revanche, elles posent des problèmes d'auto-représentation des objets, de gestion du polymorphisme ou encore d'accès aux attributs. La technique d'encapsulation de méthode nécessite une récriture globale de la classe originale mais permet de bien contrôler ces différents mécanismes. C'est donc la technique que nous utiliserons.

III.2.1.2 Java : un support d'exécution partiellement réflexif

Le langage Java fournit certaines propriétés réflexives à travers l'API Reflection et une métaclasse dénommée *java.lang.class*. L'interface de cette métaclasse permet d'obtenir le métaobjet correspondant à chaque classe du système. Ce métaobjet permet alors un certain niveau d'introspection de la classe correspondante : obtenir la liste de ses membres, en créer une instance, invoquer une de ses méthodes, etc. Cette forme de réfléxivité essentiellement structurelle (introspection) est somme toute bien trop limitée pour pouvoir nous fournir les informations dont nous avons besoin pour notre protocole.

Différentes solutions ont été imaginées afin de rendre le langage Java plus réflexif. Elles peuvent être classées en trois catégories : celles qui s'appliquent à la compilation, celles qui fonctionnent au chargement des classes et enfin celles qui sont basées sur un interpréteur spécialisé. Les différentes phases de compilation, de chargement et d'interprétation sont illustrées par la figure 21.

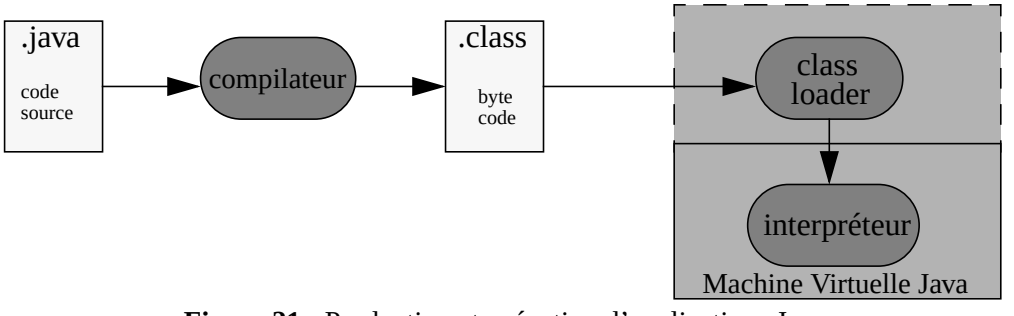

Figure 21 - Production et exécution d'applications Java

III.2.1.2.1 Réflexivité à la compilation

Dans le processus d'exécution d'un programme Java (cf figure 21), la première étape est la compilation du code source vers le code natif de la machine virtuelle, le *bytecode*. Une première solution pour rendre les objets Java réflexifs est donc d'utiliser un compilateur Java ouvert, tel qu'OpenJava [Tatsubori 1999]. OpenJava est le pendant d'OpenC++ pour le langage Java, voir I.3.2.2, page 22. Grâce à un tel outil, il est possible de modifier le code source des classes afin que les objets réifient d'eux mêmes leur comportement à l'exécution de la même manière qu'en III.2.1.1 pour le C++. Le défaut de cette approche est de nécessiter un accès au code source de toutes les classes de l'application afin de les rendre réflexives ; cela pose pro-

blème lors de l'utilisation de librairies «boîtes-noires». C'est également le cas pour les techniques présentées ci-dessus pour le langage C++, mais en Java, il est d'usage de distribuer les classes sous la forme de bytecode. Outre qu'elle permette d'obtenir des informations très fines à propos des classes compilées, cette approche a le mérite de ne nécessiter aucune modification de la machine virtuelle Java. Hormis le fait que l'on doive utiliser un compilateur spécifique, elle est donc totalement transparente.

III.2.1.2.2 Réflexivité au chargement

La machine virtuelle Java utilise une entité pour le chargement des classes (cf. figure 21). Cette entité, appelée chargeur de classe (*class loader)*, permet de ne charger en mémoire une classe qu'au moment où elle est effectivement nécessaire. De plus, le chargeur standard du langage peut etre remplacé par un chargeur personnalisé. Dalang [Welch et Stroud 1999] utilise cette fonctionnalité pour remplacer le chargeur standard par une version réflexive de celui-ci. Ce chargeur réflexif charge la classe, analyse son code (*bytecode*) et le modifie afin de rendre cette classe réflexive. Cette approche est très similaire à l'approche dite à la compilation, en effet le *bytecode* reste néanmoins un langage de haut-niveau qu'il est aisé d'analyser afin d'identifier les informations intéressantes pour implémenter un protocole à métaobjets. En revanche, la transparence de cette approche n'est que partielle, en effet, l'utilisateur doit utiliser explicitement le chargeur réflexif de Dalang sans quoi le système ne fonctionne pas.

III.2.1.2.3 Réflexivité à l'interprétation

La dernière des trois approches consiste à modifier la machine virtuelle Java et notamment son interpréteur de bytecode afin de les rendre, tous deux, réflexifs. MetaXa [Golm et Kleinoder 1999] en est un bon exemple. Grâce à cette technique il est possible d'obtenir toute sorte d'information sur le comportement des objets et celui du support d'exécution (la machine virtuelle et son environnement). On peut par exemple imaginer une machine virtuelle qui réifierait l'accès aux ressources locales telles que fichiers, sockets, tâches, etc. Le défaut de cette approche est de nuire complètement à la portabilité du système. En effet, elle requiert une machine virtuelle spécialisée qui n'est plus du tout compatible avec la norme Java. La portabilité de Java étant son plus gros avantage : «*write once, run everywhere*»[1] ; ce défaut parait rédhibitoire.

III.2.1.2.4 Java et l'information comportementale : conclusion

La réflexivité présente dans le langage Java n'est pas suffisante pour nous permettre d'implémenter des mécanismes de tolérance aux fautes. En effet, Java permet de faire de l'introspection sur les classes du système mais malheureusement pas de contrôler le comportement des objets. Nous avons présenté trois solutions pour ren-

1. «*write once, run everywhere*» (écrivez une fois, exécutez partout) est un des slogans publicitaires imaginés par Sun pour la médiatisation de son langage.

dre le langage Java plus réflexif : une solution à la compilation, une autre au chargement des classes et enfin une solution à l'exécution. Le tableau 8 présente les différents avantages et inconvénients de ces trois solutions.

Le meilleur compromis nous semble être la solution à la compilation : elle permet d'utiliser une machine virtuelle Java standard, elle est transparente, et qui plus est, elle est compatible avec la solution que nous avons présentée pour le langage C++. La solution au chargement est également intéressante car elle ne requiert pas l'accès au code source des classes de l'application. Elle peut donc être appliquée à des classes issues de librairies pour lesquelles le code source n'est pas disponible au prix d'une moins bonne transparence. La solution à l'exécution nécessite une modification de la machine virtuelle (JVM), ce qui réduit à néant sa portabilité.

	Besoin du code source	JVM d'origine	Transparence
Compilation	✔	✔	✔
Chargement		✔	
Exécution			✔

Tableau 8 - Comparaison des solutions pour rendre Java réflexif

III.2.1.3 Support d'exécution réflexif (CORBA 3.0)

Utiliser un support d'exécution réflexif est évidemment la solution la plus simple pour obtenir le type d'informations comportementales que nous désirons. Malheureusement les langages standard comme Java, C++ ou encore Ada ne fournissent pas de tels environnements d'exécution. L'OMG pour sa part désire intégrer quelques notions de réflexivité dans CORBA, une spécification d'intercepteurs est en cours d'élaboration [OMG 1998e]. Ces intercepteurs sont des objets qui se placent entre client et *stub* ou entre *skeleton* et serveur ; de nombreuses informations leurs sont réifiées, parmi celles-ci on remarquera particulièrement la création de référence, l'invocation d'une méthode par un client, la réception d'une invocation par le serveur, l'empaquetage et le dépaquetage des arguments, etc. Toutes les informations comportementales dont nous avons besoin pour implémenter la tolérance aux fautes se trouveraient aisément accessibles. Malheureusement ces intercepteurs ne sont pas encore disponibles et risquent de se faire attendre encore bien longtemps. A leur propos, on pourra se reporter aux différentes propositions faites à l'OMG, comme, par exemple [OMG 1999].

Les intercepteurs ne résolvent cependant pas tous les problèmes que nous avons énoncés, en particulier l'accès aux informations structurelles des objets (qui sont traités par la MetaObject Facility, cf I.2.2.3), à l'état interne ou externe, à l'état de l'ORB (modes de concurrence), etc. Néanmoins ces différentes spécifications montrent que le concept de réflexivité fait son chemin et ne reste pas cantonné au monde de la recherche. CORBA, qui est un outil industriel spécifié par un consortium d'industriels, commence à s'y intéresser. Dans un futur relativement proche, on

peut imaginer que l'architecture CORBA sera complètement ouverte et qu'ainsi, l'implémentation de mécanismes de tolérance aux fautes se fera plus facilement, de façon transparente et portable à la fois, à la manière dont nous le proposons ici, mais sans avoir besoin de modifier le code source des classes pour les rendre réflexives.

III.2.1.4 Obtention de l'information comportementale : conclusion

Dans cette section, nous nous sommes attachés à étudier les différents moyens d'obtenir la métainformation nécessaire aux mécanismes de tolérance aux fautes, et ce, en fonction du niveau de réflexivité offert par les couches sous-jacentes du système.

Avec un langage non réflexif, comme C++, il apparaît que la meilleure solution est d'utiliser un compilateur ouvert, tel OpenC++ v2, pour modifier le code source des classes de l'application afin de les rendre réflexives. Ces modifications peuvent prendre plusieurs formes, mais nous avons identifié une technique, l'encapsulation de méthode, qui semble être le meilleur compromis entre transparence, séparation et gestion des objets.

Avec un langage partiellement réflexif, comme Java, après avoir étudié trois différentes approches, il nous semble que la meilleure d'entre elles, du point de vue de la transparence et de la portabilité, est une approche similaire à celle décrite ci-dessus pour le langage C++. Non seulement cette approche est avantageuse de ces points de vue, mais elle permet de traiter uniformément les applications écrites dans ces deux langages, Java et C++. Etant donné les objectifs de portabilité et d'interopérabilité de langage que nous nous sommes fixés, cette approche est très séduisante.

Enfin, lorsque le support d'exécution est suffisamment réflexif pour fournir directement la métainformation désirée, il est évident que c'est la meilleure solution : transparente, portable, etc. Malheureusement, CORBA n'est pas encore prêt à fournir cette réflexivité, ce sera sans doute le cas dans quelques années.

Il est important de noter que ces approches sont compatibles entre elles : les intercepteurs CORBA sont en quelque sorte des encapsulateurs, ils pourront remplacer les mécanismes d'encapsulation générés grâce aux compilateurs ouverts et ainsi pourront collaborer avec les métaobjets déjà développés.

III.2.2 Etat des objets

Nous avons déterminé au paragraphe III.1.3 que l'état d'un objet était constitué de plusieurs types d'information :

- Une première partie qui est encapsulée dans l'objet lui-même et que l'on peut qualifier d'état interne. Cette partie est constituée de l'ensemble des attributs de l'objet.

- Une seconde partie de l'état qui, quant à elle, est externe à l'objet, et se trouve plutôt au cœur du système. Cette partie regroupe les fichiers ouverts, les tâches de l'objet, etc.

Dans cette section, nous analysons les différents moyens d'obtenir l'état des objets en fonction du niveau de réflexivité offert par le système : avec un langage partiellement réflexif comme Java, avec un langage non-réflexif comme C++ et enfin avec un système ou un ORB réflexif.

III.2.2.1 La sérialisation de Java

Depuis la version 1.1 de la spécification de Java, et l'ajout au langage des capacités d'introspection dans l'API *Reflection*, il est possible de sauvegarder l'état (interne) d'une hiérarchie d'objets dans un flot d'octets. Cette action s'appelle la sérialisation (*Serialization* [Sun 1998]).

Ce processus utilise l'introspection pour parcourir et sauvegarder les attributs d'un objet. Si un des attributs à sauvegarder est une référence sur un autre objet, ce dernier est sérialisé à son tour. Il est donc capable de sauvegarder une hiérarchie complète d'objets liés par composition.

Pour que ce processus puisse etre applicable à une classe, il suffit que cette dernière déclare implémenter l'interface *Serializable*. Elle peut alors, si elle le désire, personnaliser le processus de sauvegarde en redéfinissant les méthodes *writeObject*(), *readObject*(), *writeReplace*() et *readResolve*(), en fournissant la liste des attributs à sauvegarder dans un attribut spécial nommé *serialPersistentFields,* ou encore en déclarant les attributs à ne pas sauvegarder comme *transient.*

L'utilisation de la sérialisation est donc simple et extensible. Il suffit d'ajouter à une classe les mots clé *implements Serializable* pour que les objets issus de celle-ci puisse être sauvegardés et restaurés. L'utilisation d'un compilateur ouvert, tel qu'Open Java, permet, en outre, d'effectuer cette modification de manière transparente et automatique. Ceci montre sur un exemple simple la complémentarité de la réflexivité à la compilation et à l'exécution. Obtenir alors l'état interne d'objets Java est une chose aisée.

En ce qui concerne l'état externe des objets Java, aucune facilité n'est fournie par le support d'exécution du langage. Les tâches et les fichiers sont encapsulés dans des classes qui ne sont pas sérialisables. Il est difficile d'obtenir des informations à propos de ces entités en restant transparent et portable. En effet, les deux solutions possibles sont:

- d'une part l'utilisation de librairies spécialisées qui fournissent des classes *Thread* et *File* sérialisables. Cette solution n'est pas transparente car l'utilisateur doit explicitement utiliser les librairies fournies.
- D'autre part la modification de la machine virtuelle Java afin de pouvoir obtenir les informations nécessaires à la sauvegarde de ce type d'entités système. Cette solution nuit à la portabilité de l'application puisque celle-ci ne fonctionnerait que sur la machine virtuelle modifiée (cf. III.2.1.2.3).

On notera, toutefois, que de récents travaux proposent d'utiliser la réflexivité à la compilation afin de permettre la sérialisation d'objets Java multi-tâches [Sekiguchi *et al.* 1999].

III.2.2.2 Sauvegarde et restauration d'objets C++

Le langage C++ ne fournit pas de fonction équivalente à la sérialisation de Java, et encore moins de mécanisme d'introspection. Ce manque de réfléxivité à l'exécution peut etre compensé par l'utilisation d'un compilateur réflexif afin d'implémenter une forme de sérialisation pour les objets C++.

En effet, lorsque l'introspection peut être effectuée au moment de la compilation, il est alors possible de générer des méthodes *Base_SaveState*() et *Base_RestoreState*(). Ces méthodes, respectivement, sauvegardent et restaurent l'ensemble des attributs d'un objet, de façon équivalente aux méthodes *writeObject* et *readObject* de Java. Cette approche est utilisée dans deux études différentes [Kasbekar *et al.* 1999] et [Killijian *et al.* 1999].

De plus, nous proposons d'utiliser la réflexivité à la compilation pour optimiser la taille de l'état sauvegardé [Killijian *et al.* 1999] et [Ruiz-García *et al.* 1998]. En effet, grâce à OpenC++ v2, il est possible de modifier le code source des méthodes, afin que lors de l'exécution, les attributs modifiés soient identifiés. De cette manière, nous pouvons fournir à l'objet une méthode *Base_SavePartialState()* qui ne sauvegarde que les attributs précédemment modifiés. Cette méthode réinitialise tous les attributs comme non modifiés, ainsi lors de l'appel suivant, elle ne fournira que les attributs qui ont été modifiés entre temps. Une méthode *Base_RestorePartialState()* est également fournie pour l'interprétation et la restauration de cette liste partielle d'attributs.

Obtenir l'état interne d'objets C++ est donc chose possible grâce à l'utilisation d'un compilateur réflexif. Tout comme c'était le cas pour le langage Java, obtenir l'état externe des objets est bien plus difficile. Les solutions sont les mêmes et reposent également sur des techniques non transparentes ou non portables :

- Des librairies spécialisées qui gèrent les éléments de l'état externe des objets et qui permettent une sauvegarde et une restauration de ceux-ci. L'utilisateur doit explicitement utiliser les classes fournies par ces librairies.
- Des appels à des fonctions internes du système d'exploitation. Par exemple, le système d'exploitation Solaris permet d'obtenir certaines informations concernant les tâches ou les fichiers. Ces informations sont évidemment de très bas niveau et sont donc difficilement exploitables.

En ce qui concerne la gestion des fichiers et d'autres ressources du même type, nous proposons une approche intermédiaire qui, certes, réduit la transparence mais a le mérite d'être portable et de permettre une sauvegarde et une restauration de l'état de ces ressources. Cette approche consiste à regrouper la gestion de ces ressources dans des serveurs dédiés, un serveur de fichier par exemple. L'application

utilise alors ce serveur pour accéder à ces ressources. La technique de sérialisation présentée ci-dessus est alors appliquée aux serveurs afin de sauvegarder et de restaurer leur état.

III.2.2.3 La réflexivité, un concept transversal

Nous désirons implémenter des mécanismes pour assurer de façon transparente des propriétés non-fonctionnelles au sein d'applications à base d'objets CORBA. Afin d'assurer la transparence de ces mécanismes du point de vue du programmeur de l'application, nous proposons de les implémenter au sein de métaobjets. Ces métaobjets ont besoin, pour remplir correctement leur tâche, de pouvoir (cf. figure 22) :

- Observer et contrôler le comportement des objets de l'application,
- Sauvegarder et restaurer l'état interne de ces objets et,
- Dans la mesure du possible, sauvegarder et restaurer l'état externe des objets qui réside dans le support d'exécution ou dans le système d'exploitation.

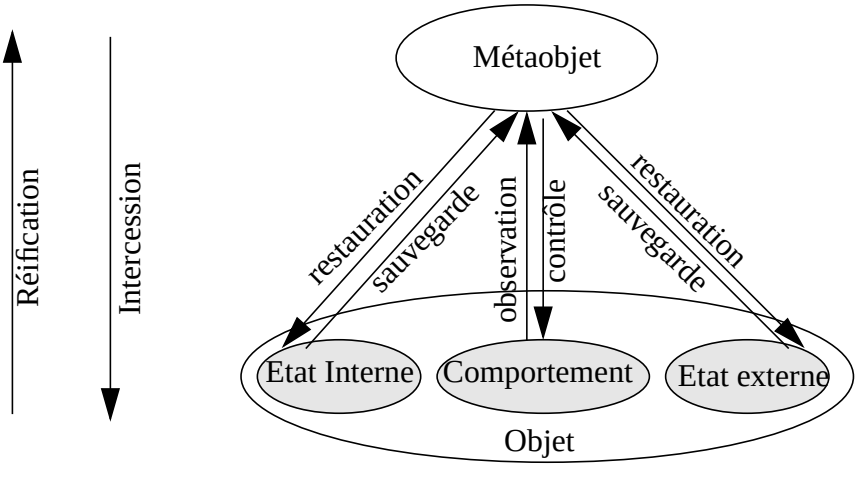

Figure 22 - Relations entre Objet et Métaobjet

Nous venons de voir lors des deux précédents paragraphes qu'obtenir l'état interne des objets était possible grâce à la réfléxivité du langage, que ce soit au moment de l'exécution ou de la compilation. Observer et contrôler le comportement des objets est également possible grâce à une combinaison de ces deux approches. Pour sa part, l'état externe des objets est constitué d'informations système qui sont difficilement accessibles d'une manière portable.

Tout comme la réfléxivité à la compilation et à l'exécution sont complémentaires pour obtenir l'état interne des objets ainsi que pour observer et contrôler leur comportement, la réfléxivité du système d'exploitation ou du support d'exécution (cf. figure 23) serait d'un grand secours pour obtenir l'état externe de ces mêmes objets.

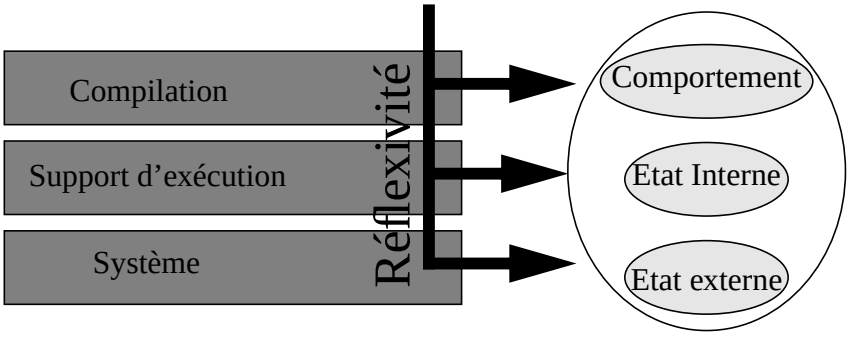

Figure 23 - Utilisation de la réflexivité à tous les niveaux

Malheureusement, les systèmes d'exploitation réflexifs ne sont pas légion à l'heure actuelle et sont encore moins standardisés, nous en avons cité quelques-un en III.1.3.2. Etant donné que nous désirons nous abstraire du système d'exploitation, l'utilisation de l'un d'entre eux n'est pas acceptable. Une manière élégante de résoudre ce problème serait d'utiliser les propriétés réflexives d'un système abstrait, tel que Posix ou d'une sur-couche au système telle que CORBA, pour obtenir ces informations. Malheureusement ni l'un, ni l'autre, n'est suffisamment ouvert pour proposer de telles informations.

Plusieurs équipes de recherche sont actuellement lancées dans l'étude d'ORBs réflexifs. Par exemple, un ORB réflexif est proposé dans [Blair *et al.* 1998], dans celui-ci, trois modèles se partagent les méta-informations :

- Le méta-espace *composition*, qui regroupe les informations concernant les liens entre les objets du système.
- Le méta-espace *encapsulation*, qui fournit les informations à propos des attributs et méthodes d'un objet.
- Le méta-espace *environnement*, qui, pour sa part, exhibe les informations concernant l'ORB à proprement parler. Ces informations vont de l'arrivée des requêtes à l'encodage des paramètres (à la manière des intercepteurs de l'OMG) en passant par la création et le contrôle des tâches.

L'utilisation du méta-espace environnement nous permettrait de contrôler, par exemple, l'état externe des objets multi-tâches, et donc de pouvoir sauvegarder et restaurer de tels objets.

III.2.2.4 Etat des objets : conclusion

Nous avons étudié dans cette section différents moyens d'obtention de l'état des objets. L'état d'un objet est composé de deux parties : la première, dite interne, est encapsulée dans l'objet lui-même et est constituée de l'ensemble des valeurs de ses attributs ; la seconde, dite externe, réside dans les différentes couches du système et est constituée de différentes informations difficiles à identifier comme les fichiers ouverts, les tâches, etc.

En ce qui concerne l'état interne des objets, plusieurs solutions existent. La première solution est fournie par le langage Java. En effet, ce langage, bien que peu réflexif, fournit un niveau d'introspection suffisant pour pouvoir sauvegarder et restaurer l'état interne des objets. Il permet même de personnaliser les méthodes responsables de la gestion de l'état pour chaque classe. Une deuxième solution consiste à employer un compilateur réflexif pour générer de telles méthodes de gestion de l'état des objets, cette solution est valable pour des langages non-réflexifs, comme C++ par exemple.

Il est intéressant de noter que la norme CORBA commence à s'intéresser également à la possibilité de transférer l'état des objets. C'est le cas avec la dernière version de la norme qui défini une interface pour le transfert d'objets par valeur (*Object by Value*) et non plus par référence, ou encore du service de persistance des objets. Ces différentes caractéristiques n'étant que des interfaces, obtenir et restaurer l'état des objets est toujours d'actualité.

Pour ce qui est de l'état externe des objets, il n'existe pas, à l'heure actuelle, de solution satisfaisante. En effet, plusieurs travaux de recherche portent sur les systèmes ouverts et notamment les ORBs réflexifs. Grâce à de tels outils il sera possible d'obtenir cet état externe d'une façon encadrée par le protocole fourni. En attendant que ces recherches portent leurs fruits, quelques solutions ponctuelles existent. On peut, par exemple, regrouper la gestion des fichiers dans un serveur, éventuellement répliqué. Ce serveur sera rendu tolérant aux fautes soit par journalisation, soit en gérant un état interne qui permette de restaurer l'état des fichiers ouverts. Cet état serait, par exemple, constitué des noms des fichiers ainsi que de la position de l'index courant en lecture ou en écriture.

III.2.3 Conclusion

Dans cette section, nous avons examiné différentes solutions pour obtenir la méta-information nécessaire à la réalisation de mécanismes de tolérance aux fautes. Cette méta-information est constituée de deux parties : la première concerne le comportement des objets et la seconde leur état. Les différentes solutions diffèrent selon le niveau de réfléxivité du langage et du système utilisé, elles sont résumées dans le tableau 9 et expliquées ci-dessous.

	Comportement	Etat interne	Etat externe
Java	compilateur ouvert ou ClassLoader réflexif ou machine virtuelle spécifique	✔	
C++	compilateur ouvert	compilateur ouvert	
Intercepteurs CORBA	✔		
ORB ouvert	✔	✔	
Système ouvert			✔

Tableau 9 - Obtention de la méta-information : différentes solutions

Pour un langage non-réflexif comme C++, la solution consiste en l'utilisation d'un compilateur réflexif afin de modifier le code source des classes de l'application. Ces modifications ont deux buts distincts : premièrement la réification et l'intercession du comportement des objets et deuxièmement de permettre la sauvegarde et la restauration de leur état.

Avec un langage partiellement réflexif, comme Java, on peut utiliser les mécanismes du langage permettant la sauvegarde et la restauration de l'état des objets. Ces mécanismes sont rendus possibles grâce, notamment, à la réflexivité du langage. En ce qui concerne le comportement des objets, le langage lui-même ne propose rien. Il existe néanmoins différentes solutions selon que l'on accepte d'utiliser une machine virtuelle spécifique, un système peu transparent, ou que l'on utilise un compilateur ouvert. Afin de permettre l'interopérabilité des langages Java et C++, nous utiliserons cette dernière solution. Elle permet notamment d'utiliser le même protocole à métaobjets pour des objets C++ et Java. Ainsi, un objet Java peut être contrôlé par un métaobjet écrit en C++, ou vice-versa.

Dans un futur relativement proche, des ORBs et des systèmes ouverts devraient voir le jour. D'ailleurs ce processus a déjà commencé avec l'introspection dans Java ou la MetaObject Facility et les intercepteurs dans CORBA. En se basant sur les documents disponibles, on peut imaginer que ceux-ci permettront non seulement la réification du comportement des objets mais également l'obtention de leur état externe.

III.3 Définition du protocole à métaobjets

A partir des informations identifiées dans les sections précédentes, il nous est maintenant possible de définir clairement le protocole à métaobjets que nous désirons implémenter. Ce protocole fait intervenir plusieurs entités, voir figure 24. Au premier plan sont le client et le serveur. Le client communique avec le serveur à travers

un stub. Nous avons vu dans les sections précédentes qu'il était nécessaire de contrôler les invocations sortantes du client ainsi que les références détenues par celui-ci. C'est le rôle du métastub. Enfin le serveur est contrôlé par son métaobjet.

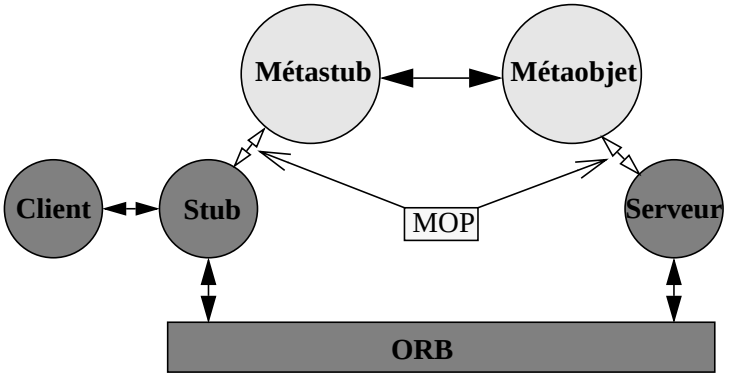

Figure 24 - Les intervenants du protocole à métaobjets

Le protocole à métaobjets régit les interactions entre ces différentes entités : entre le stub et son métastub, entre l'objet et son métaobjet et enfin entre le métastub et le métaobjet. Ces différentes entités sont des objets CORBA et, par conséquent, communiquent entre elles à travers des interfaces IDL. Nous définirons donc le protocole à métaobjets en le décrivant sous la forme des interfaces IDL des différentes entités.

III.3.1 Interface des métaobjets

La principale activité d'un métaobjet est, comme nous l'avons étudié jusqu'à présent, l'observation et le contrôle de l'objet qui lui est associé. Pour ce faire le métaobjet dispose des deux outils issus de la réflexivité : la réification et l'intercession.

III.3.1.1 Réification

La réification correspond à l'interception des évènements que subit l'objet, et à leur transmission au métaobjet. Cette transmission se fait à travers un appel de méthode. L'objet appelle une méthode du métaobjet pour lui réifier l'évènement qu'il vient de recevoir. Les méthodes qui concernent cette partie du protocole à métaobjets, la réification, font donc partie de l'interface du métaobjet.

Du point de vue de l'interception, l'objet doit réifier au métaobjet les évènements suivants:

- **La création de l'objet**, c'est à dire l'appel au constructeur. Selon le langage, la classe peut éventuellement définir plusieurs constructeurs. Lors de la réification de la création de l'objet, on doit fournir au métaobjet l'identification du constructeur ainsi que ses paramètres d'appel. La méthode du métaobjet qui réifie l'appel au constructeur est :

```
void Meta_StartUp(in long constuctorID, inout any arguments);
```

- **L'invocation entrante d'une méthode** : chaque invocation par un client d'une méthode de l'interface IDL de l'objet doit être réifiée. Cette réification doit fournir l'identificateur de la méthode appelée ainsi que ses paramètres. La méthode du métaobjet correspondante à cette réification est :

```
void Meta_MethodCall(in long methodID, inout any arguments);
```

- **La destruction de l'objet** : selon le langage utilisé, la sémantique de la destruction de l'objet diffère. En C++ par exemple, un seul destructeur peut être défini par classe; il n'a aucun paramètre; l'appel à celui-ci implique que l'objet sera effectivement détruit après son exécution ; il peut être appelé explicitement ou le sera éventuellement par le support d'exécution du langage. En Java, par contre, la notion de destructeur n'existe pas vraiment. La machine virtuelle implémente un ramasse-miette qui détruit éventuellement les objets identifiés comme inutiles grâce au comptage de références. En cas de destruction, le ramasse-miette appelle la méthode *Finalize*. Cette méthode peut donc être considérée comme le destructeur de l'objet à ceci près qu'elle ne peut pas être appelée explicitement et pourra éventuellement ne jamais être appelée. En tout état de cause, dans ces deux langages il n'existe qu'un destructeur par classe et celui-ci n'a aucun paramètre. L'appel à celui-ci sera donc réifié simplement sans identification ni paramètres par l'appel de la méthode

```
void Meta_CleanUp();
```

III.3.1.2 Interface pour la gestion du lien métaobjet-objet

Au delà de ses fonctions de contrôle et d'observation, le métaobjet fait partie d'un ensemble d'entités, appelé communément le métaniveau. Ce métaniveau rassemble les différents métaobjets et métastubs du système ainsi qu'une autre entité, la fabrique de métaobjets. Cette dernière est chargée de la création, de la destruction, et de la modification dynamique des métaobjets pour le compte des objets.

Hormis les méthodes nécessaires à l'interception, le métaobjet devra implémenter certaines méthodes spécifiques à la gestion du méta-niveau. Pour pouvoir modifier dynamiquement le métaobjet associé à un objet, la fabrique de métaobjets a besoin de pouvoir **connaître et** éventuellement **modifier l'objet associé au métaobjet**. Pour ce faire, le métaobjet doit définir les deux méthodes suivantes :

```
Object Meta_GetObject();
void Meta_SetObject(in Object newObject);
```

Il nous est dorénavant possible de donner l'interface IDL correspondante pour un métaobjet, cf. tableau 10.

```
interface Metaobject {
// Méthodes pour la réification
      void Meta_StartUp   ( in long constuctorID,
                            inout any arguments);
      void Meta_MethodCall( in long methodID,
                            inout any arguments);
      void Meta_CleanUp();
// Méthodes pour la gestion du métaniveau
      Object Meta_GetObject();
      void Meta_SetObject(in Object newObject);
};
```

Tableau 10 - Interface IDL des métaobjets

III.3.2 Interface des objets

Tout comme la réification prend la forme d'un appel de méthode de l'objet vers le métaobjet, l'intercession, ou le contrôle, est effectué par l'invocation par le métaobjet d'une méthode de l'objet.

III.3.2.1 Intercession

En ce qui concerne le contrôle de l'objet par le métaobjet, on distingue les activités suivantes :

- **L'obtention de l'état de l'objet** : le métaobjet doit pouvoir obtenir l'état de l'objet d'une manière indépendante de la classe de celui-ci. L'état de l'objet doit donc être fourni encapsulé dans une structure générique. Dans ce but, l'objet fournit la méthode suivante :

```
any Base_SaveState();
```

- **La restauration de l'état de l'objet** : parallèlement à son obtention, le métaobjet doit pouvoir fournir un nouvel état à l'objet en appelant la méthode :

```
void Base_RestoreState(in any newState);
```

- **Activation par le métaobjet d'une méthode** (constructeur, méthode, destructeur) : le métaobjet doit pouvoir invoquer un constructeur, une méthode ou le destructeur. Pour les deux premiers il doit identifier la méthode particulière que sa requête concerne et fournir ses arguments d'appel ; pour le destructeur aucune information supplémentaire n'est nécessaire. Les méthodes de l'objet qui correspondent respectivement à ces actions sont :

```
void Base_StartUp   (in long constructorID, inout any arguments);
void Base_MethodCall(in long methodID, inout any arguments);
void Base_CleanUp();
```

III.3.2.2 Méthodes pour la gestion du lien avec métaniveau

Comme nous l'avons indiqué précédemment, la fabrique de métaobjets est responsable de la création et de la destruction des métaobjets pour le compte des objets. Elle est aussi garante des liens qui unissent objets et métaobjets, elle sert de médiateur entre ces deux types d'entités. Elle peut avoir besoin de **changer dynamiquement le métaobjet associé à un objet**, en cas de reconfiguration du système par exemple. Pour le permettre, l'objet devra implémenter les deux méthodes suivantes :

```
Metaobject Base_GetMetaobject();
void Base_SetMetaobject(in Metaobject newMetaobject);
```

Par conséquent, l'interface IDL d'un objet peut être définie par le tableau 11.

III.3.2.3 Interface des stubs et métastubs

Les stubs effectuent les invocations sortantes d'un objet. Il est nécessaire de réifier ces invocations pour la plupart des mécanismes de tolérance aux fautes. Ils sont également détenteurs des références sur le serveur. Il est donc également nécessaire de contrôler ces références pour qu'une défaillance du serveur soit masquée à ses clients. En revanche, mis à part la référence qu'il détient, l'état du stub ne nous intéresse pas, il ne dépend que de l'implémentation de l'ORB.

```
interface ReifiedObject {
// Méthodes pour la gestion de l'état
      any Base_SaveState();
      void Base_RestoreState(in any newState);
//Méthodes pour l'intercession
      void Base_StartUp   ( in long constructorID,
                            inout any arguments);
      void Base_MethodCall( in long methodID,
                            inout any arguments);
      void Base_CleanUp();
// Méthodes de gestion du métaniveau
      Metaobject Base_GetMetaobject();
      void Base_SetMetaobject(in Metaobject newMetaobject);
};
```

Tableau 11 - Interface IDL des objets

Les interfaces des stubs et métastubs sont donc en corrélation directe avec ces deux types d'information : invocation sortante et référence.

Le tableau 12 donne l'interface des stubs. Celle-ci correspond dans l'ensemble à l'interface d'un objet sans les méthodes de gestion de l'état. Les méthodes *Base_Stub_GetReference* et *Base_Stub_SetReference* permettent au métastub de

contrôler la référence que détient le stub sur l'objet cible, cette référence est du type générique *Object* afin de pouvoir s'adapter à n'importe quel objet. Les méthodes *Base_GetMetastub* et *Base_SetMetastub* permettent de gérer le lien stub-métastub.

```
interface ReifiedStub {
// Gestion du comportement et des invocations
  void Base_Stub_StartUp   ( in long constructorID,
                             inout any arguments);
  void Base_Stub_MethodCall( in long methodID,
                             inout any arguments);
  void Base_Stub_CleanUp();
// Gestion des references
  Object Base_Stub_GetReference();
  void Base_Stub_SetReference(in Object newReference);
  Metastub Base_GetMetastub();
  void Base_SetMetastub(in Metastub newMetastub);
};
```

Tableau 12 - Interface IDL des Stubs

Enfin, le tableau 13 décrit l'interface des métastubs. Elle est plus ou moins équivalente à celle d'un métaobjet. Les méthodes *Meta_GetStub* et *Meta_SetStub* permettent de gérer le lien métastub-stub.

```
interface Metastub {
// Gestion du comportement
  void Meta_Stub_StartUp     ( in long constructorID,
                               inout any arguments);
  void Method_Stub_MethodCall( in long methodID,
                               inout any arguments);
  void Meta_Stub_CleanUp();
// Interface pour la gestion du metaniveau
  ReifiedStub Meta_GetStub();
  void Meta_SetStub(in ReifiedStub newStub);
};
```

Tableau 13 - Interface IDL des Métastubs

III.3.3 Empaquetage et dépaquetage des arguments

La réification et l'intercession des appels de méthode, de constructeur ou de destructeur, nécessite la fourniture au métaniveau des arguments de ces appels. Ces arguments doivent être encapsulés dans une structure générique pour que, quels que soient leur nombre et leurs types, on puisse les fournir aux méthodes de réification et d'intercession.

L'action consistant à générer une telle structure à partir des arguments est l'**empaquetage**, et l'action inverse, qui consiste à générer les arguments à partir de cette structure, est le **dépaquetage**.

Le standard CORBA définit un type de donnée générique qui peut contenir n'importe quel autre type de donnée, qu'il soit simple ou composite (structures, unions, etc.). C'est ce type de donnée *any*, que nous utilisons pour transmettre les arguments aux méthodes *Meta_StartUp*, *Meta_MethodCall (cf. III.3.1.1)*, *Base_StartUp* et *Base_MethodCall* (cf. III.3.2.1). De plus le compilateur IDL permet de générer automatiquement des fonctions d'empaquetage et de dépaquetage pour une structure de donnée définie en IDL (cf. tableau 14).

Structure IDL	Code d'empaquetage/dépaquetage généré par le compilateur IDL

`structure Simple {`
`  long a;`
`  String b;`
`};`
 	
`structure Complexe`
`{`
`  Simple compose;`
`  double g;`
`};`
	`// Paquetage : Simple vers any`
	`void operator <<=(CORBA_Any&, Simple_ptr*);`
	`// Depaquetage : any vers Simple`
	`CORBA_Boolean operator >>=`
	`(const CORBA_Any&, Simple_ptr&);`
	
	`// Paquetage : Complexe vers any`
	`void operator <<= (CORBA_Any&, Complexe_ptr);`
	`// Depaquetage : any vers Complexe`
	`CORBA_Boolean operator >>=`
	`(const CORBA_Any&, Complexe_ptr&);`

Tableau 14 - Génération automatique des fonctions d'empaquetage/dépaquetage par le compilateur IDL

III.4 Conclusion

Dans ce chapitre, nous avons défini un protocole à métaobjets qui permet d'implémenter des protocoles de tolérance aux fautes. Ce protocole à métaobjets fournit essentiellement trois types d'informations. Premièrement, il permet au métaobjet de contrôler le comportement de l'objet en termes de création, invocation de méthodes et destruction. Deuxièmement, il donne les moyens au métaobjet de sauvegarder et de restaurer l'état des objets. Et enfin, il permet une gestion dynamique du lien entre objets et métaobjets.

L'information relative au comportement des objets et à son contrôle sera disponible au niveau de l'ORB lorsque la spécification CORBA 3 sera prête. En effet, il est prévu que cette norme inclue des intercepteurs qui peuvent se placer entre, d'une part, client et stub, et d'autre part, skeleton et serveur. Des informations comme la création de référence ou l'invocation de méthode, seront réifiées à ces intercepteurs. Pour l'instant, une autre solution est l'utilisation de langages réflexifs pour instrumenter le code des classes de l'application. Cette instrumentation permet la réification des informations comportementales nécessaires à un métaobjet.

L'état d'un objet est, lui-même, constitué de deux parties distinctes. La première partie, l'état interne, est constituée de l'ensemble des attributs de l'objet, qu'ils soient de type simple, de type composé ou encore d'objets. La deuxième partie, l'état externe, est composée d'informations liées à l'objet, mais qui résident, non pas au sein de l'objet lui-même, mais au cœur du système d'exploitation ou de l'ORB.

Si grâce à un langage réflexif on peut obtenir suffisamment d'information pour générer des méthodes de sauvegarde et de restauration de l'état interne des objets, ce n'est pas le cas pour l'état externe. Obtenir ou restaurer l'état externe des objets implique d'utiliser une plate-forme -système d'exploitation et ORB- réflexive.

Se basant sur les abstractions fournies par CORBA, notre protocole à métaobjets s'intègre parfaitement dans cette architecture. En effet, non seulement les métaobjets sont des objets CORBA, mais ils utilisent les outils fournis par celui-ci pour le contrôle du comportement et de l'état des objets.

L'utilisation de la réflexivité à la compilation permet donc d'implémenter le protocole à métaobjets que nous venons de définir, et ce, au dessus d'une plate-forme COTS et en utilisant des langages différents, ici C++ ou Java. Cette implémentation fait l'objet du chapitre suivant.

CHAPITRE IV
IMPLÉMENTATION DU PROTOCOLE À MÉTAOBJETS

IV.1 Introduction

Au chapitre 3, nous avons défini un protocole à métaobjets à l'exécution pour la sûreté de fonctionnement des systèmes répartis à objets CORBA. Nous avons également observé que, de manière générale, les langages et plates-formes logicielles (systèmes et ORBs) utilisés pour le développement de systèmes répartis n'étaient pas assez ouverts pour permettre l'implémentation d'un tel protocole dans tous ses aspects et toute sa généralité.

Dans ce chapitre, nous proposons d'utiliser un langage réflexif, en l'occurrence OpenC++, pour implémenter de façon transparente ce protocole à métaobjet. En effet, ni C++ (ou même Java), ni CORBA, ne nous permettent d'obtenir les informations nécessaires à sa mise en œuvre. La solution que nous proposons est d'utiliser OpenC++, et son compilateur ouvert, pour identifier ces informations lors de la compilation, et permettre de les exhiber au moment de l'exécution [Killijian *et al.* 1998a],[Killijian *et al.* 1998b]. Nous montrons que la réflexivité au niveau langage est un atout majeur pour le développement de protocoles à métaobjets spécifiques sur des supports d'exécution non réflexifs.

De plus, l'utilisation d'un tel compilateur permet, non seulement d'ajouter de façon transparente le protocole à métaobjets aux objets du système distribué, mais également de vérifier que certaines règles de programmation sont respectées. En effet, ces règles doivent être incontournables pour assurer la cohérence des stratégies de tolérance aux fautes appliquées. Or, le langage C++ est un langage hybride entre langages procéduraux, comme le langage C, et langages orientés-objet. Certaines des facilités que propose le C++ au programmeur sont hautement indésirables du point de vue des mécanismes de tolérance aux fautes. OpenC++ nous permet alors de filtrer les programmes afin d'identifier l'utilisation de telles fonctionnalités et d'aider le programmeur à les éradiquer.
L'approche présentée dans ce chapitre est également appliquée, avec des adaptations mineures, au langage Java en utilisant le compilateur réflexif d'OpenJava.

IV.1.1 Approche

L'approche consiste à rendre réflexives les classes d'une application distribuée qui utilise CORBA. Cette modification, qui correspond à l'ajout du protocole à métaobjets défini au chapitre précédent, doit être, autant que faire se peut, transparente pour le développeur de l'application. L'utilisation d'un compilateur réflexif permet justement d'analyser et de modifier, de façon transparente, le code source des classes d'une application. L'unique contrainte du développeur est d'utiliser le compilateur réflexif d'OpenC++ en lieu et place de son compilateur habituel.

Le compilateur réflexif associe une métaclasse à chaque classe de l'application (cf. figure 25). Cette métaclasse est responsable de l'analyse et de la traduction de la classe qui lui est associée. Pour implémenter notre protocole à métaobjet, nous devons donc écrire une métaclasse qui sera chargée d'ajouter les propriétés de réification et d'intercession de notre protocole aux classes de l'application. Le compilateur génère alors une nouvelle classe qui correspond à la classe originale modifiée par sa métaclasse. Cette nouvelle classe peut finalement être compilée avec un compilateur C++ standard, ce processus est illustré figure 25.

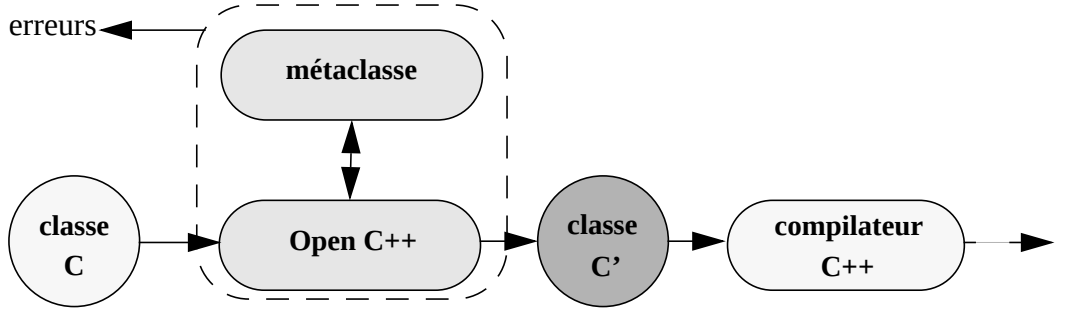

Figure 25 - Processus de compilation

Comme tout compilateur, l'ensemble compilateur réflexif et métaclasse peut identifier et renvoyer des erreurs à l'utilisateur. Ces erreurs peuvent être des erreurs de programmation du langage C++ mais également des erreurs détectées pendant l'analyse effectuée par la métaclasse. Ce dernier point est très important puisqu'il permet à la métaclasse de définir de nouveaux types d'erreurs et, suivant son analyse, de filtrer ces erreurs dans les classes dont elle est responsable. Nous reviendrons sur cette possibilité en section IV.2.

IV.1.2 Fonctionnement d'OpenC++ v2

Avant de rentrer dans les détails de l'implémentation de notre protocole à métaobjet, il est nécessaire d'avoir une relative connaissance du mode de fonctionnement du compilateur réflexif qu'est OpenC++ v2.

Comme nous l'avons brièvement décrit précédemment, le compilateur d'OpenC++ associe une métaclasse à chaque classe de l'application. Cette métaclasse peut être soit la métaclasse par défaut *Class*, qui génère une classe (*C'*) équivalente à la

classe d'entrée (*C*), soit une métaclasse particulière, qui utilise les outils fournis par OpenC++ pour analyser la classe d'entrée et générer une classe de sortie différente. Cette métaclasse personnalisée hérite de la métaclasse *Class* et, ainsi, se comporte par défaut comme cette dernière. Elle redéfinit alors les méthodes dont elle désire modifier le comportement. Les méthodes de la métaclasse *Class* que nous utiliserons principalement sont les suivantes :

- *TranslateClass* : cette méthode, qui est la méthode principale, est responsable de l'analyse de la partie déclarative de la classe d'entrée. C'est dans cette méthode que l'on peut analyser et modifier la liste des membres (méthodes et attributs) ou la hiérarchie d'héritage de la classe d'entrée.
- *TranslateMemberFunction* : cette méthode est responsable de l'analyse du code d'une méthode membre de la classe d'entrée. Elle est appelée par OpenC++ pour chaque définition de méthode. C'est dans cette dernière que l'on peut notamment modifier l'implémentation des méthodes et analyser les paramètres d'entrée.
- *TranslateMemberRead* et *TranslateMemberWrite* : ces deux méthodes sont appelées à chaque fois qu'un attribut est respectivement lu ou écrit. C'est à partir des informations fournies par celles-ci que l'on peut mettre en place les techniques de capture d'état partiel et d'accesseurs (pour la notion d'accesseur, cf. IV.2.1.3 et IV.3.3).

Dans la section suivante, nous montrons comment utiliser un compilateur réflexif, tel que celui d'OpenC++, pour l'application de conventions de programmations. Comme nous le verrons, ce processus de filtrage du code source de l'application est parfois nécessaire.

IV.2 Filtrage du code

La réalisation d'un protocole à métaobjets s'appuyant sur la réflexivité à la compilation dépend de certaines caractéristiques fondamentales du langage à objet utilisé. Elle repose sur un modèle à objet «propre» qui doit être imposé. En effet, les langages de programmation sont généralement conçus avec un compromis entre un modèle objet figé et une souplesse d'utilisation :

- Le langage C++ est plus souple, voire permissif, que d'autres, comme Java par exemple. En effet, le C++ est un descendant direct du langage C, qui, lui-même est très permissif. Il est par exemple aisé, avec le C++, de ne pas respecter le principe d'encapsulation. Il en résulte, du point de vue de la sûreté de fonctionnement, que le langage C++ possède un modèle à objets faible et permet trop de souplesse.
- Java, quant à lui, utilise un modèle à objets mieux défini. Il autorise néanmoins certaines constructions, relatives à l'encapsulation, que nous ne pouvons tolérer dans le contexte de la sûreté de fonctionnement.

Dans cette section, nous analysons les caractéristiques des langages considérés (C++ et Java) et nous montrons comment la réfléxivité à la compilation permet de garantir un modèle de programmation adéquat. Enfin, nous conclurons en indiquant les nombreux bénéfices d'un tel filtrage, y compris dans d'autres contextes que celui de la tolérance aux fautes.

IV.2.1 Adéquation du modèle objet de C++

Le langage C++ est réputé pour être un langage hybride (entre orienté-objet et procédural) et donc permissif. En effet, bien qu'il soit orienté-objet, il hérite de bien des aspects de son ancêtre le langage C. Ce dernier langage a été conçu dans le but de pouvoir inter-agir facilement, et de façon efficace, avec le système d'exploitation et le matériel sous-jacent à l'application. Ces objectifs se sont traduits par une très grande souplesse du langage quant à l'accès aux données.

Premièrement, une encapsulation faible découle directement de cette souplesse. Une classe peut déclarer la totalité ou certains de ses attributs comme non soumis à l'encapsulation, ainsi d'autres classes peuvent y accéder.

Deuxièmement, le plus gros défaut de C++, qui est sujet à de multiples controverses, est constitué par ses pointeurs. Les pointeurs en C++ sont les plus souples qu'il soit : à travers un pointeur on peut accéder à n'importe quelle partie de la mémoire de l'application, et ce, d'une manière totalement incontrôlée. L'utilisation des pointeurs permet non seulement de briser l'encapsulation mais, comme nous le verrons, pose d'autres problèmes majeurs.

IV.2.1.1 Encapsulation

L'encapsulation est une propriété très importante dans notre contexte. On veut, en effet, pouvoir contrôler l'état des objets et, pour ce faire, celui-ci doit être encapsulé dans l'objet lui-même. On veut éviter qu'un objet quelconque vienne modifier intempestivement l'état de l'objet que nous contrôlons. C'est d'autant plus important lorsque nous nous intéressons à l'état partiel de l'objet. En effet, l'état partiel est construit à partir des modifications de l'état contrôlées par l'objet lui-même. Si ces modifications ne peuvent être contrôlées, on ne peut identifier les attributs modifiés, et on ne peut donc pas identifier l'état partiel de l'objet.

IV.2.1.1.1 Protection des attributs

En C++, les attributs et méthodes sont qualifiés par un type d'accès. Trois types d'accès sont possibles: *public*, *private* et *protected*. Le tableau 15 résume ces différentes informations.

- L'accès *public* spécifie qu'un objet quelconque peut accéder en lecture et en écriture à l'attribut en question, ou, s'il s'agit d'une méthode, qu'elle est visible et utilisable par tout objet.
- L'accès *private* est l'inverse de *public* : aucun objet, quelle que soit sa classe, ne peut accéder à l'attribut ou à la méthode ainsi qualifié.

- Enfin le type d'accès *protected* est un compromis entre les deux autres : le membre (attribut ou méthode) est considéré comme *public* pour les objets issus de sous-classes de la classe courante, et il est considéré comme *private* pour les autres.

Au delà de la visibilité, ces types d'accès influent implicitement sur l'héritage : membres public et protected sont hérités puisque visibles aux sous-classes alors que les membres privés ne sont pas hérités puisqu'invisibles.

	Objet lui même	Objet d'une sous-classe	Objet de classe quelconque
Public	✔	✔	✔
Protected	✔	✔	
Private	✔		

Tableau 15 - Types d'accès aux membres d'une classe

Du point de vue de la cohérence de l'état, un accès en lecture aux attributs, quels qu'ils soient, n'est pas gênant. En revanche l'accès en écriture de manière incontrôlée, i.e. qui ne passe pas par une méthode que nous contrôlons, ne permet alors plus d'assurer cette cohérence. Deux solutions s'offrent alors : la première, radicale, consiste à interdire purement et simplement les attributs *public* et *protected* ; la seconde consiste à remplacer les accès directs aux attributs par des appels de méthodes. Cette dernière solution est bien plus souple et transparente que la première car elle est moins restrictive et permet une solution automatisée ; nous la présentons en détail au paragraphe IV.2.1.3.

IV.2.1.1.2 Fonctions et classes amies

Un deuxième aspect qui peut briser l'encapsulation en C++ est la déclaration de fonctions ou de classes *amies* (avec le mot clé *friend*). Dans la définition d'une première classe, déclarer une seconde classe comme étant amie permet aux méthodes de cette dernière d'accéder à tous les attributs de la première classe, y compris aux attributs privés. Déclarer une fonction amie permet la même chose, mais à cette fonction uniquement.

La raison invoquée pour la nécessité de ce type de déclaration est l'efficacité. Si les fonctions (ou classes) amies n'existaient pas, une classe qui aurait besoin d'accéder à des attributs d'une autre classe devrait invoquer une méthode à chaque accès. Si ces accès sont nombreux, comme pour un calcul matriciel par exemple, le temps d'exécution serait rédhibitoire [Stroustrup 1992].

A notre avis, cet argument, s'il était valable en 1991 lors de la conception du langage, ne l'est plus aujourd'hui. En effet, les compilateurs ont fait d'énormes progrès dans l'optimisation du code qu'ils produisent. Il est très courant que ceux-ci remplacent un appel de méthode par le code de la méthode elle-même lorsque celui-ci est relativement court. On appelle cette technique les méthodes "en ligne". Il est

également possible de forcer le compilateur à mettre “en ligne” une méthode, en utilisant le mot-clé *inline*. Evidemment, une méthode qui se contente de renvoyer la valeur d’un attribut est excessivement courte. Avec les compilateurs d’aujourd’hui, si le mode optimisé est sélectionné, un tel appel est toujours remplacé par le code de la méthode.

Par conséquent, il ne nous paraît pas que le fait d’interdire de telles classes et fonctions amies soit trop limitant pour l’utilisateur. Si toutefois c’était le cas, il serait éventuellement possible de n’autoriser que les accès aux attributs en lecture à ce types de classes et fonctions.

IV.2.1.1.3 Variables globales et variables de classe

Malgré le fait qu’il soit un langage orienté-objet, le langage C++ permet toutefois de déclarer des variables et fonctions globales. Ces variables et fonctions sont alors partagées par toute l’application, c’est-à-dire, par tous les objets de l’application. Encore une fois, l’écriture dans de telles variables partagées est à proscrire. En effet, si un objet écrit une certaine valeur dans une telle variable, celle-ci doit être considérée comme faisant partie de l’état de l’objet. Ceci est surtout vrai si l’objet se réfère à cette valeur dans le futur ; dans le cas contraire, la variable est utilisée comme canal de communication caché entre différents objets, ce qui n’est pas non plus acceptable pour la sûreté de fonctionnement, au sens large. On notera que ces variables globales n’existent pas avec le langage Java pour qui tout est un objet (cf. IV.2.2).

Dans le modèle à objet de C++ (et dans celui de Java), il existe également des variables de classe (déclarées avec le mot clé *static*). Ces variables sont alors partagées entre toutes les instances de la classe, ce qui pose le même problème que pour les variables globales. On peut éventuellement tolérer la lecture de telles variables, mais il est difficile de s’accommoder d’un usage en écriture.

IV.2.1.2 Les pointeurs

Comme nous l’avons dit en I.1.2.2, un pointeur a, de manière générale, un rôle équivalent à une référence, à savoir la désignation d’un objet. En C++, toutefois, un pointeur peut être utilisé pour désigner un emplacement mémoire quelconque (un objet, un attribut, une variable, une méthode, une fonction ou encore un emplacement non-alloué). Bien qu’il puisse référencer tout type d’emplacement, le pointeur C++ est typé, ce qui implique que l’emplacement désigné par un pointeur est accédé en fonction du type du pointeur. Un pointeur C++ correspond finalement à une adresse mémoire typée.

Ce système de désignation est très souple, et correspond assez bien à la philosophie du langage : souplesse et efficacité. Malheureusement, cette souplesse se traduit par la possibilité donnée au programmeur de faire, littéralement, n’importe quoi à partir d’un pointeur : accéder aux données privées d’un objet, accéder à un nombre réel comme s’il était une chaîne de caractères, etc. Ceci augmente de façon incommensurable la probabilité d’introduire des erreurs dans l’application.

IV.2.1.2.1 Arithmétique de pointeur

Dans son optique de souplesse, le C++ permet d'utiliser l'arithmétique de pointeurs : la valeur d'un pointeur (l'adresse qu'il contient) peut être le résultat d'une opération arithmétique.

L'avantage de cette possibilité est de permettre d'implémenter aisément et efficacement des tables de hachage, par exemple. Une table de hachage est un tableau, à priori relativement creux, dans lequel sont classées des données. La position d'une donnée dans le tableau est déterminée par une fonction (la fonction de hachage) paramétrée par la donnée elle-même. Une fonction de hachage peut être par exemple : f(entier)=entier . Le tableau16 donne l'exemple du parcours d'un tableau avec une telle fonction de hachage, ce parcours peut-être effectué soit avec un pointeur, soit de manière plus classique.

Parcours avec pointeur	Parcours standard

`void parcours(int Entier) {`
`  // Fonction de hachage`
`  int iDebut=Entier^2;`
`  int iFin=(Entier+1)^2;`
`  int * pEntier; // Le pointeur`
`  for (i=iDebut;i<iFin;i++) {`
`  pEntier = adresseTableau+i;`
`  *pEntier = Entier;`
`  }`
`}`
	`void parcours(int Entier) {`
	`  // Fonction de hachage`
	`  int iDebut=Entier^2;`
	`  int iFin=(Entier+1)^2;`
	`  for (i=iDebut;i<iFin;i++) {`
	`  Tableau[i] = Entier;`
	`  }`
	`}`

Tableau 16 - Parcours de tableau avec ou sans arithmétique de pointeur

A la lecture de ces deux fonctions, il apparaît clairement que la méthode utilisant le pointeur pour l'accès au tableau est bien moins lisible que celle qui accède à celui-ci simplement avec un indice. Pour cet exemple simple, le parcours par pointeur n'est pas plus efficace que le parcours simple, mais avec des fonctions de hachage plus complexes, ce pourrait être le cas.

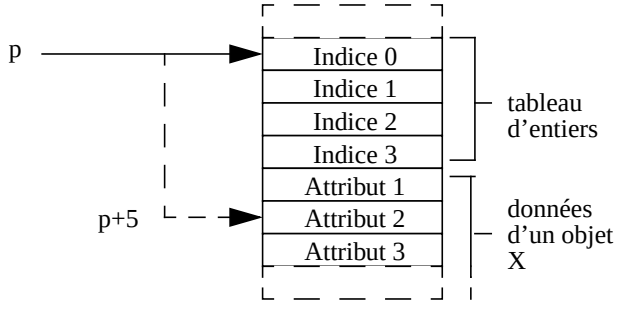

Figure 26 - Accès aux données privées par arithmétique de pointeurs

Ce léger avantage de l'arithmétique de pointeur est à contrebalancer avec ses nombreux inconvénients. Premièrement, l'écriture condensée induit un manque de lisibilité indéniable et mène, comme nous l'avons dit, à l'introduction d'erreurs. Ces erreurs peuvent conduire facilement à dépasser le cadre du tableau et par exemple à recouvrir une portion mémoire appartenant à un autre objet. Deuxièmement, et plus pernicieusement, cela permet également d'accéder intentionnellement à des données privées d'un objet, cf. figure 26. Dans cet exemple simple, l'arithmétique de pointeurs est utilisée sur le pointeur *p* (*p+5*) pour accéder aux attributs d'un autre objet.

Qu'ils soient intentionnels ou pas, nous ne pouvons permettre ces accès intempestifs aux données d'un objet. De plus, comme nous le verrons plus tard dans ce chapitre, de tels accès ne peuvent être contrôlés à la compilation, et donc rendent impossible la technique d'obtention de l'état partiel des objets que nous utilisons.

IV.2.1.2.2 Pointeurs dans les appels de fonctions ou de méthodes

Il est courant en C++ d'utiliser des pointeurs comme paramètres de méthodes ou de fonctions. En effet, le langage C n'offrait pas de système de références, et, bien que le C++ ait intégré un tel système, la coutume d'utiliser les pointeurs est restée.

Pour une méthode, les paramètres peuvent être utilisés en d'entrée, en sortie, ou encore en entrée-sortie. Etant donné la technique qu'emploie le langage pour le passage des paramètres, à savoir un passage par valeur, les paramètres destinés à la sortie doivent être passés par adresse, c'est-à-dire par pointeur ou par référence, le tableau 17 donne un exemple de chacun de ces types de passage et leurs effets sur le paramètre effectif.

Paramètres par valeur	Paramètres par pointeur	Paramètres par référence

`void foo(int v) {`
`  v=3;`
`}`
`void exemple() {`
`  int local=0;`
`  foo(local);`
`  // local=0`
`}`
	`void foo(int *v) {`
	`  *v=3;`
	`}`
	`void exemple() {`
	`  int local=0;`
	`  foo(&local);`
	`  // local=3`
	`}`
		`void foo(int &v) {`
		`  v=3;`
		`}`
		`void exemple() {`
		`  int local=0;`
		`  foo(local);`
		`  // local=3`
		`}`

Tableau 17 - Les différents types de passage de paramètre

Le passage de paramètre par valeur ne permet donc pas d'utiliser le paramètre en sortie, ce que permet un passage par référence. Le passage de paramètre par pointeur permet d'une part d'utiliser ce paramètre en sortie, et, d'autre part, de conserver un pointeur sur ce paramètre pour l'utiliser utltérieurement. Cette dernière possibilité est très gênante : à tout moment, il est possible que l'objet (ou la fonction) à qui l'on a fourni le pointeur, se serve de celui-ci pour écrire à nouveau dans la variable (ou l'attribut) désigné.

Cette fonctionnalité permet donc également de briser l'encapsulation, et ce, de manière non contrôlable dans le temps. Elle est donc indésirable car elle n'est pas compatible avec notre technique d'obtention de l'état.

IV.2.1.3 Conclusion et propositions de solution

Le langage C++ a été, dès son origine, conçu dans une optique de souplesse et d'efficacité. Cette souplesse, malheureusement, est obtenue au détriment de la rigueur du système de typage et du principe d'encapsulation. Bien des fonctionnalités sont donc, de ces points de vue, indésirables : les attributs publics, les classes et fonctions amies, les variable globales et de classe et, enfin, les nombreuses fonctionnalités attribuées aux pointeurs.

Pour améliorer la confiance que nous plaçons dans les applications, nous proposons d'utiliser la réflexivité à la compilation, pour filtrer ces fonctionnalités non désirées, les réifier à l'utilisateur et lui proposer une alternative (alternative qui d'ailleurs existe toujours).

Pour ce qui est de l'efficacité, nous pensons que les compilateurs actuels, avec leurs capacités d'analyse et d'optimisation, permettent d'obtenir dans la majorité des cas des résultats équivalents. De toute façon, dans une optique de sûreté de fonctionnement, l'efficacité est à placer au second plan, c'est la confiance et la rigueur du modèle de programmation qui priment.

En ce qui concerne les accès aux attributs, nous proposons la solution décrite au paragraphe suivant. Pour les autres fonctionnalités, Open C++ sera utilisé pour identifier leur usage et l'interdire (avec un message d'erreur). Il peut éventuellement être envisagé d'utiliser OpenC++ pour remplacer automatiquement ces fonctionnalités par leur équivalent correct.

Pour pallier aux désagréments évoqués dans cette section, et en particulier au paragraphe IV.2.1.1.1, nous proposons, grâce à OpenC++, de générer automatiquement des fonctions d'accès aux attributs. A chaque attribut de chaque classe sont alors associés trois *accesseurs*. Ces accesseurs offrent la même visibilité que les attributs dont ils sont responsables: accesseurs publics pour les attributs publics, etc.

Ces trois accesseurs sont:

- un accesseur en lecture, qui renvoie la valeur de l'attribut en retour,
- un accesseur en lecture/écriture préfixée qui écrit la nouvelle valeur de l'attribut puis la renvoie,
- un accesseur en lecture/écriture postfixée qui écrit la nouvelle valeur de l'attribut mais renvoie l'ancienne valeur.

Grâce à ces trois accesseurs, il est possible de remplacer tous les accès directs aux attributs par des appels de méthodes. De cette manière il est beaucoup plus aisé de contrôler l'accès aux attributs. De plus, les modifications d'attributs sont alors regroupées dans les deux accesseurs de lecture/écriture. Ainsi les modifications du code de l'application nécessaires à la gestion de l'état partiel des objets sont limitées à ces deux méthodes.

On donne un exemple de tels accesseurs dans le tableau 18. Dans cet exemple, la classe *Exemple* définit deux attributs: *iPub*, qui est public et *fPriv* qui est privé. La méthode *foo* utilise la valeur de l'un pour affecter l'autre puis inversement. Dans la deuxième colonne, on voit le code des accesseurs : trois accesseurs privés pour l'attribut privé *fPriv,* ainsi que trois accesseurs publics pour l'attribut public *iPub.* Le code de la méthode *foo* a été modifié pour utiliser les accesseurs.

Code original	Code avec les accesseurs

`class Exemple {`
`public:`
`  int iPub;`
`private:`
`  float fPriv;`
`...`
`  void foo() {`
`  ...`
`  iPub=(int)fPriv+4;`
`  ...`
`  fPriv=(float)iPub++;`
`  }`
`  ...`
`};`
	`class Exemple {`
	`public:`
	`  int iPubLect() {return iPub;}`
	`  int iPubPrefix(int i) {`
	`  iPub=i;return iPub;}`
	`  int iPubPostfix(int i) {`
	`  int t=iPub;iPub=i;return t;}`
	`private:`
	`  int iPub;`
	`  float fPriv;`
	`  float fPrivLect() {`
	`  return fPriv;}`
	`  ... // Deux autres accesseurs`
	`...`
	`  void foo() {`
	`  ...`
	`  iPubPrefix((int)fPrivLect()+4);`
	`  ...`
	`  fPrivPrefix((float)iPubPostfix(`
	`  iPubLect()+1));`
	`  }`
	`...`
	`};`

Tableau 18 - Exemples d'utilisation des accesseurs

Lorsque l'attribut n'est pas accédé directement, mais par le biais d'un paramètre de fonction, ces accesseurs ne suffisent pas en eux-mêmes. Il faut alors, en plus, modifier le code de l'appel de fonction afin qu'il utilise explicitement les accesseurs. Nous donnons un exemple au tableau 19.

Dans cet exemple, on part d'un code original avec une classe qui possède un attribut entier public *i* ainsi qu'une méthode *foo* qui prend en paramètre un entier par référence et qui modifie la valeur de ce paramètre. Dans une autre méthode (*bar*), on appelle la méthode *foo* avec l'attribut *i* en paramètre, de telle sorte que cet attribut en retour soit modifié. Dans le code avec accesseurs, donné dans la deuxième colonne, on voit qu'afin d'utiliser les accesseurs, l'appel de la méthode *foo* a été remplacé par trois lignes de code : la première définit une variable temporaire et lui

affecte la valeur de l'attribut *i*, la seconde effectue l'appel à *foo* avec cette variable temporaire en paramètre, enfin, la troisième ligne copie la valeur de la variable temporaire dans l'attribut *i*.

Code original	Code avec accesseurs

`class Exemple {`
`public:`
`  int i;`
`  ....`
`};`
 	
`void Exemple::foo(int &p) {`
`   p=12;`
`   ...`
`}`
`void Exemple::bar() {`
`   foo(i);`
`   ...`
`}
	`class Exemple {`
	`private:`
	`   int i;`
	`public:`
	`   iLect....`
	`   iEcriPrefix...`
	`   iEcriPostfix...`
	`...`
	`};`
	`void Exemple::foo(int &p) {`
	`   p=12;`
	`   ...`
	`}`
	`void Exemple::bar() {`
	`   int tempo=iLect();`
	`   foo(tempo);`
	`   iEcriPrefix(tempo);`
	`   ...`
	`}

Tableau 19 - Appels de fonctions et accesseurs

De telles modifications concernant l'usage d'attributs dans les paramètres des appels de méthode ou de fonction sont nécessaires car la lecture ou l'écriture de l'attribut ne peut pas être identifiée lors de la compilation. En effet, le corps de la méthode *foo*, seul, ne permet pas de savoir si le paramètre est un attribut ou non ; être capable d'identifier un tel fonctionnement requièrerait une analyse du code qui sortirait du champ de cette thèse. En revanche, dans le corps de la méthode *bar*, on se rend compte qu'un attribut est utilisé comme paramètre d'une méthode et que ce paramètre étant passé par référence, l'attribut est susceptible d'être modifié. On peut donc modifier le code de l'appel pour qu'il utilise les accesseurs.

Ces modifications peuvent paraître complexes pour un simple appel de méthode, mais elles reflètent les opérations implicites qui se déroulent lors d'un appel de méthode avec un paramètre en entrée/sortie : premièrement la valeur du paramètre est lue, puis, l'appel est effectué avec cette valeur en paramètre et enfin la valeur de retour est écrite dans le paramètre. On notera que l'utilisateur n'a pas à se soucier de ce type d'utilisation puisque les modifications de son code sont effectuées automatiquement.

Ce qui est intéressant avec cette technique, c'est qu'elle nécessite uniquement la modification de la méthode appelante et non de la méthode appelée. Elle peut donc fonctionner avec des fonctions dont on ne connaît que l'en-tête et pas l'implémentation, comme c'est le cas avec les librairies de classes.

IV.2.2 Adéquation du modèle objet de Java

Java est un langage bien plus propre, au sens de son modèle à objet, que ne l'est C++. Le premier de ses avantages est de définir un modèle de référence clair : il n'existe pas de pointeurs en Java. Les références ne sont utilisées que pour accéder aux objets, l'arithmétique ne s'y applique pas.

En revanche, au niveau de la visibilité des attributs, Java propose un modèle plus riche, et par là même plus complexe, que celui de C++ : en plus des types de visibilité *public*, *protected* et *private*, Java propose un nouveau type, le type *default*. Ce type de visibilité concerne les classes définies dans le même paquetage que la classe courante. Un paquetage est constitué d'un ensemble de classes. Tous les membres non-privés sont visibles par les classes d'un même paquetage. En effet, les concepteurs du langage ont considéré qu'un paquetage est l'œuvre d'une seule personne ou d'un seul groupe de personnes et que, par conséquent, les classes d'un même paquetage peuvent "se faire confiance". Le tableau 20 récapitule les différentes visibilités de ces types d'accès.

	public	default	protected	private
non sous-classes du même paquetage	✔	✔	✔	
sous-classes du même paquetage	✔	✔	✔	
non sous-classes d'un paquetage différent	✔			
sous-classes d'un paquetage différent	✔		✔	

Tableau 20 - Visibilité des membres de classe Java

Tout comme en C++, les types de visibilité affectent également l'héritage, mais cette fois encore de manière plus complexe : un attribut à visibilité *default* sera hérité seulement par les sous-classes du même paquetage, les autres règles sont identiques à celles du C++ et ne concernent pas les paquetages.

Les arguments que nous avons opposés à ces pratiques au paragraphe IV.2.1.1.1 sont tout aussi valables pour Java qu'ils l'étaient pour C++. En conséquence, nous proposons de générer également pour Java des accesseurs pour chaque attribut, de la même façon que nous l'avons proposé pour C++.

Enfin, Java ne définit pas explicitement de classes amies ; en effet, la notion de paquetage et son implication au niveau de la visibilité des membres fait que toutes les classes d'un même paquetage peuvent être considérées comme amies les unes des autres.

Clairement, grâce à son modèle à objets plus propre, Java nécessite bien moins de filtrage que C++. Dans notre contexte, l'utilisation d'accesseurs permet de s'affranchir de toute fonctionnalité gênante dans le contrôle des objets par leur métaobjet.

IV.2.3 Filtrage dans d'autres contextes

Les capacités d'analyse et de filtrage de compilateurs réflexifs comme OpenC++ ou OpenJava sont utilisées ici dans un contexte de sûreté de fonctionnement, pour augmenter la confiance que l'on porte dans un langage de programmation et, par conséquent, dans les applications qui l'utilisent pour leur implémentation.

Néanmoins, il existe d'autres domaines dans lesquels de telles capacités peuvent être utiles. C'est, par exemple, le cas de l'application de conventions de programmation ou de la vérification logicielle, le test.

IV.2.3.1 Conventions de programmation et langages spécifiques

Dans certains domaines industriels, où le système développé est hautement critique en terme de vies humaines, comme l'avionique ou le nucléaire, il existe des conventions de programmation que chaque développeur doit respecter. Ces conventions spécifient des fonctionnalités du langage à ne pas utiliser, ou encore des métriques concernant quelque attribut du code source de l'application. Par exemple, il arrive souvent, dans ces domaines, que toutes les variables doivent être allouées statiquement, ou encore que le polymorphisme soit interdit. Dans de tels cas, un compilateur réflexif peut être utilisé pour vérifier ces conventions de programmation de la même façon que nous l'utilisons pour interdire l'arithmétique de pointeurs, par exemple.

En généralisant cette approche, qui consiste en fait à réduire un langage pour son utilisation dans un contexte particulier, on peut dire que celle-ci pourrait être utilisée pour l'implémentation de langages dédiés (Domain Specific Languages-DSL). Ces langages dédiés sont des langages de programmation à l'expressivité réduite qui permettent, pour un domaine spécifique, une utilisation plus simple, la vérification de certaines propriétés, ou encore une meilleure efficacité [Thibault 1998], [Consel et Marlet 1998].

IV.2.3.2 Vérification d'hypothèses pour le test logiciel orienté-objet

Dans le domaine du test orienté-objet, on s'attache à tester particulièrement quatre grandes fonctionnalités du modèle objet : l'héritage, la composition, la délégation et le polymorphisme. Il n'existe pas, à notre connaissance, de méthodologie générale de test qui permette de prendre en compte l'ensemble de ces quatre composantes. Les solutions générales, qui proposent de fournir des ordres de test ou d'aider à la définition de jeux de tests, font donc des hypothèses concernant l'utilisation ou la non-utilisation par l'application de ces fonctionnalités. Pour le test du polymorphisme, notamment, il n'existe à l'heure actuelle que des solutions au cas par cas.

Le filtrage, tel que nous le présentons dans cette section, peut alors être utilisé pour la vérification des hypothèses de la méthodologie de test utilisée. Par exemple, on pourra vérifier qu'aucune classe de l'application que l'on désire tester, n'utilise le polymorphisme. Après quoi, on peut, en toute confiance, appliquer la méthodologie proprement dite.

IV.2.4 Conclusion

Nous avons vu dans cette section, que bien souvent, un langage de programmation comme C++ ou Java pouvait être trop général, ou trop souple, pour certaines utilisations, ou pour certains domaines d'utilisation. C'est notamment le cas pour la sûreté de fonctionnement, par exemple dans le test logiciel, ou encore lorsque dans un contexte industriel on doit assurer que certaines règles de programmation sont respectées, pour les applications critiques.

Dans ces différents cas, la compilation réflexive permet d'une part d'analyser le code source des applications de façon cohérente et d'autre part, d'attirer l'attention de l'utilisateur sur certains points de l'application qui peuvent poser problème. Nous appelons ce processus le *filtrage*, et nous étudions, ici, tout particulièrement l'adaptation des langages C++ et Java au développement d'applications distribuées à objets rendues tolérantes aux fautes de façon transparente grâce au protocole à métaobjets que nous développons dans cette thèse.

IV.3 Instrumentation des classes

Nous présentons en détail, dans cette section, la solution qui permet de rendre réflexives des classes d'une application distribuée à objets CORBA. Le protocole à métaobjets ajouté aux objets de ces classes est celui que nous avons défini au chapitre III.

IV.3.1 Introduction

Avec le protocole à métaobjets que nous avons défini, les intervenants de la communication entre un client et un serveur sont nombreux (cf. figure 27) :

- Le client du serveur, il utilise un stub particulier afin de réifier les invocations sortantes.
- Le stub spécifique à notre protocole, il implémente l'interface *ReifiedStub*, destinée aux interactions avec son métastub.
- Le métastub, il implémente l'interface *Metastub*, est responsable du fonctionnement du stub et inter-agit avec le métaobjet.
- Le métaobjet du serveur, implémente l'interface *Metaobject*, il contrôle le comportement et l'état du serveur grâce au protocole à métaobjet.

- Le serveur à proprement parler, il implémente l'interface IDL du service qu'il fournit à son ou ses clients, et implémente également l'interface *ReifiedObject* pour permettre à son métaobjet de le contrôler.
- Et enfin, la fabrique de métaobjet (interface *MOFactory*) qui crée métaobjets et métastubs, à la demande, respectivement, du serveur et du stub, et qui permet à ces deux entités de se découvrir afin de communiquer entre elles.

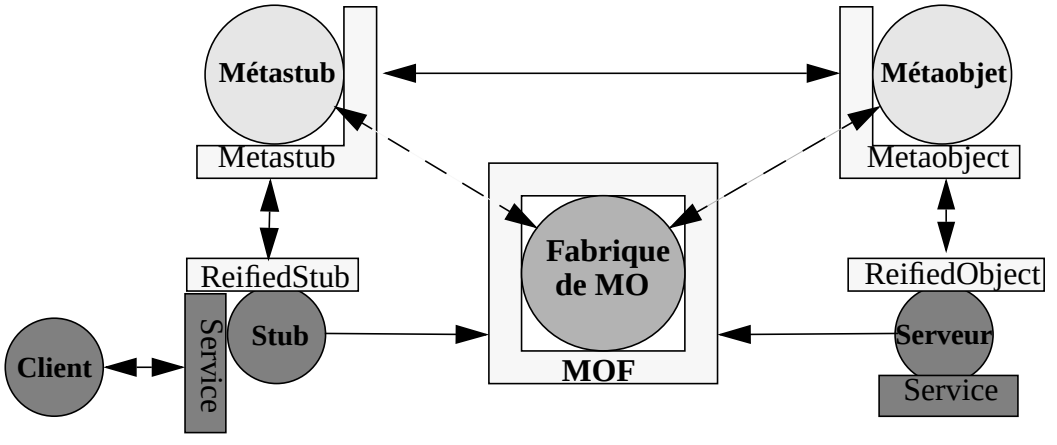

Figure 27 - Intervenants et Interfaces du Protocole à Métaobjets

Parmi ces intervenants, seul le serveur doit être modifié par notre compilateur réflexif. En effet, métaobjet, métastub et fabrique de métaobjet sont développés par le responsable du métaniveau, le stub peut être généré à partir du code source du serveur et, enfin, le client ne subit aucune modification (sauf si d'un autre côté il est également serveur).

Nous présentons donc, dans cette section, la métaclasse utilisée pour l'analyse et la modification du code source de la hiérarchie de classes du serveur, afin, d'une part, de le rendre réflexif, et d'autre part, de générer le stub réflexif qu'utiliseront ses clients.

IV.3.2 Comportement des objets

La première fonction du protocole à métaobjets est le contrôle du comportement des objets. Ce contrôle consiste, d'une part, à réifier la création de l'objet, l'invocation des méthodes et la destruction de l'objet, puis d'autre part, à permettre l'intercession de ces évènements, c'est-à-dire à permettre au métaobjet de réaliser effectivement la création et la destruction d'une instance ainsi que l'invocation d'une méthode.

Pour chacune de ces fonctionnalités, il est nécessaire de prendre en compte les paramètres des méthodes invoquées, qu'elles soient un constructeur ou une simple méthode. Nous présentons dans le paragraphe suivant la technique utilisée pour l'empaquetage et le dépaquetage de ces arguments.

IV.3.2.1 Empaquetage et dépaquetage des arguments

Nous avons présenté en III.3.3 (page 77) la justification d'un tel empaquetage et dépaquetage des arguments. Brièvement, les divers constructeurs et méthodes d'une classe ont des arguments différents en nombre et en types. Pour la réification et l'intercession, il est nécessaire de ne pas dépendre des signatures de ces différents membres de la classe. Il faut donc pouvoir empaqueter et dépaqueter les arguments d'un appel dans une structure générique.

Les arguments des constructeurs et des méthodes d'une classe sont d'un des types suivants :

- un type de base du langage : *int, float, char, string (char*),* etc.
- un type composé du langage : *union, struct,* tableau, etc.
- un objet.

Chacun des types de base et composé du langage C++ a une correspondance en langage IDL. Ces diverses correspondances sont spécifiées par [OMG 1998a] (cf. tableau 21). En ce qui concerne les objets, nous utilisons notre technique d'obtention de l'état des objets, qui fournit un *any* correspondant à l'état de l'objet. Le type IDL correspondant à un objet est donc le type *any.*

type C++	type IDL
`int`	`long`
`float`	`float`
`char`	`char`
`char*`	`string`
`class x`	`any`

Tableau 21 - Correspondance des paramètres entre C++ et IDL (exemples)

Le compilateur IDL est capable de générer des fonctions d'empaquetage et de dépaquetage entre une structure IDL et le type *any.* Afin d'utiliser cette capacité, il nous faut donc générer une structure IDL qui correspond aux différents arguments de chaque méthode (un exemple est fourni au tableau 22). La méthode *MakeIDL* est responsable de la génération de cette structure IDL et de sa compilation par le compilateur IDL. *MakeIDL* est appelée par la méthode *TranslateClass* pour chaque méthode publique de la classe d'entrée. *MakeIDL* analyse alors chacun des arguments de la méthode et génère la structure IDL correspondante.

Méthode Source en C++	Structure Correspondante en IDL

`long FonctionF(`
`  long l,`
`  float f,`
`  char* s,`
`  char c,`
`  UneClasse &o`
`  );`
	`struct FonctionF_arguments {`
	`  long result;`
	`  long l;`
	`  float f;`
	`  string s;`
	`  char c;`
	`  any o;`
	`};`

Tableau 22 - Structure IDL générée pour l'empaquetage et le dépaquetage des arguments d'une méthode

IV.3.2.2 Réification

Nous avons présenté en III.2.1.1 (page 58) les différentes techniques d'encapsulation qui permettaient de réifier les appels de méthodes d'une classe C++. Nous y avons également justifié notre choix pour la technique d'encapsulation de méthode plutôt que celle d'encapsulation de classe. Voici comment nous réalisons ce mécanisme avec OpenC++.

Chaque fonction membre (constructeur, méthode ou destructeur) est tout d'abord renommée. Pour ce faire, la fonction *TranslateClass* définit un nouveau membre avec les mêmes arguments que la méthode originale et un nom différent, par exemple *Real_foo* pour la méthode *foo*. Ensuite la méthode *TranslateMemberFunction* renomme l'implémentation de la méthode en question et crée une nouvelle méthode responsable de la réification (la méthode d'encapsulation à proprement parler). Cette nouvelle méthode possède le nom original de la méthode ainsi que les mêmes arguments. Son implémentation est triviale en ce qui concerne les méthodes : il s'agit d'empaqueter les arguments, d'appeler la méthode *MetaMethodCall* du métaobjet et de dépaqueter les arguments et le paramètre de retour.

Pour ce qui est des constructeurs et du destructeur, la nouvelle méthode a un comportement plus complexe.

Le constructeur de l'objet doit tout d'abord initialiser l'ORB et créer son *skeleton* correspondant à l'interface *ReifiedObject*. Ensuite il doit se mettre en relation avec la fabrique de métaobjet et lui demander de créer son métaobjet. Une fois qu'il a obtenu une référence sur son métaobjet, il doit invoquer la méthode *MetaStartup* de celui-ci. Cette dernière action nécessite donc d'empaqueter les arguments du constructeur, d'invoquer cette méthode et de dépaqueter les arguments.

Le destructeur de l'objet correspond à la finalisation du cycle de vie du serveur. Il doit donc tout d'abord invoquer la méthode *MetaCleanup* de son métaobjet, demander à la fabrique de métaobjet la destruction de son métaobjet et enfin s'extraire de l'ORB.

La méthode *TranslateMemberFunction* génère ces différentes méthodes d'encapsulation pour chaque méthode publique de la classe d'entrée. Pour ce faire, elle analyse donc le membre et ses paramètres.

Dans le chapitre III, nous avons justifié le fait que les appels internes de l'objet sur lui-même ne devaient pas être réifiés. Il faut donc traduire tous les appels de méthode interne afin que pour chacun d'entre eux, l'objet appelle la méthode originale qui a été renommée. La méthode *TranslateMemberCall* de la métaclasse permet de traduire les appels aux méthodes de la classe d'entrée. Ainsi, il est aisé de remplacer les appels internes dans le code des méthodes originales : il suffit de remplacer le nom de la méthode appelée par son nouveau nom, *Real_methode*.

IV.3.2.3 Intercession

Pour que le contrôle du comportement de l'objet soit complet, le métaobjet doit pouvoir activer les méthodes originales de l'objet (*Real_*). Le protocole à métaobjets le permet par l'intermédiaire de la méthode *Base_HandleCall*. Celle-ci doit dépaqueter les arguments de l'appel, réaliser l'invocation de la méthode encapsulée et empaqueter les paramètres ainsi que la valeur de retour. L'implémentation de cette méthode dépend donc de chacune des méthodes originales de la classe d'entrée. Elle est donc générée par la méthode *TranslateClass*, qui est la seule à connaître chacun des membres de la classe d'entrée. Pour chacun de ses membres, elle doit générer le code de dépaquetage, créer une variable temporaire pour le retour éventuel de la méthode originale ainsi que le code d'empaquetage de ses arguments de retour.

En ce qui concerne les constructeurs, l'approche est rigoureusement identique, la méthode *Base_StartUp* est équivalente à *Base_HandleCall* : en fonction du constructeur désiré, elle dépaquète les arguments, réalise l'appel et empaquète les arguments.

La méthode *Base_CleanUp* est sensiblement plus simple puisqu'un seul destructeur existe et qu'il n'a ni argument ni valeur de retour. Elle se contente donc de permettre l'accès à celui-ci.

IV.3.2.4 Le comportement des objets en Java

En ce qui concerne la gestion du comportement des objets, Java n'est pas très différent de C++. L'unique différence concerne les destructeurs. En Java, le destructeur d'une classe est la méthode *finalization* alors qu'en C++, celui-ci porte le nom de la classe (*~Exemple* pour la classe *Exemple*). La métaclasse OpenJava, utilisée pour rendre réflexives les classes Java, a donc un fonctionnement très proche de celui présenté dans cette section pour OpenC++.

IV.3.3 Etat interne des objets

Au paragraphe III.2.2.2 (page 68), nous avons exposé une technique pour implémenter la sérialisation d'objets C++ grâce à la réflexivité à la compilation. Nous présentons maintenant sa réalisation en deux parties : tout d'abord comment générer des méthodes pour la sauvegarde et la restauration de l'état interne complet des objets, puis, ensuite, deux autres méthodes qui permettent d'obtenir et de restaurer l'état interne partiel des objets.

On notera que ces deux techniques ne s'appliquent qu'à l'état interne des objets. Pour une description plus détaillée sur les différences entre état interne et état externe, ainsi que pour les différentes techniques d'obtention de l'état externe, on se reportera au chapitre précédent.

IV.3.3.1 Etat interne complet

L'état interne complet d'un objet est défini à un instant donné par les valeurs de l'ensemble de ses attributs. Bien que l'état soit dynamique, c'est-à-dire qu'il varie dans le temps, la liste des attributs est définie statiquement par la classe de l'objet. On peut donc générer des méthodes qui sauvegardent ou restaurent l'état de l'objet dans, ou à partir, d'une structure définie statiquement.

Tout comme c'était le cas pour les paramètres d'appel de méthode, l'état de l'objet doit être disponible sous une forme générique. En effet, les prototypes des méthodes de gestion de l'état ne varient pas en fonction de la classe de l'objet. Les fonctions d'empaquetage et dépaquetage de l'état font donc l'objet du paragraphe suivant.

IV.3.3.1.1 Empaquetage et dépaquetage

Pour l'empaquetage et le dépaquetage de l'état, nous utilisons la même technique que pour les paramètres des méthodes. La métaclasse génère, à la compilation, une structure IDL qui permet de stocker l'état des objets de la classe d'entrée, un exemple est fourni au tableau 23. La correspondance entre le type C++ des attributs de la classe et le type IDL de la structure est essentiellement la même qu'au paragraphe IV.3.2.1.

classe d'entrée	structure IDL

`class Mere {`
`  private:`
`   char* Init;`
`   int nb;`
`   Mere* suivant;`
`};`
`class Exemple :`
` private Mere {`
`  private:`
`  int x,y;`
`};`
	`struct MereState {`
	`  String Init;`
	`  int nb;`
	`  any suivant;`
	`};`
	
	`class ExempleState {`
	`  any Mere;`
	`  int x;`
	`  int y;`
	`};`

Tableau 23 - Structure IDL pour l'empaquetage et le dépaquetage de l'état des objets

Seule la gestion des objets est différente : si un des attributs est un pointeur sur un objet, l'objet pointé peut exister ou ne pas exister. S'il existe, on est capable d'obtenir son état par un appel à la méthode *Base_SaveState* (cf. IV.3.3.2.3), mais s'il n'existe pas, cette information doit être sauvegardée afin de pouvoir être restaurée correctement (attribut pointant dans le vide, sur *null*). De plus, il se peut que plusieurs pointeurs désignent le même objet, c'est souvent le cas dans des structures de données complexes comme des arbres ou des listes chaînées. Ces informations sont capitales afin de pouvoir restaurer l'état correct de l'objet. Lorsque l'on se trouve face à un pointeur sur un objet, plusieurs cas de figure sont alors possibles :

- L'objet désigné existe et n'a pas encore été sauvegardé. On doit alors obtenir son état et lui assigner une référence[1]. Il faut stocker dans la structure cet état ainsi que la référence.
- L'objet désigné existe mais a déjà été sauvegardé. On doit alors retrouver sous quelle référence il a été précédemment sauvegardé. Il faut stocker la référence dans la structure.
- L'objet désigné n'existe pas. Il n'a donc, a fortiori pas d'état. Il faut indiquer que le pointeur a donc une valeur nulle non significative.

Afin de représenter ces différents cas de figure dans la structure IDL qui correspond à l'état de l'objet, un type IDL particulier est défini ; ce type est présenté au tableau 24. Une première structure, *PtrAndState* permet de stocker l'état de l'objet ainsi que la référence qu'on lui a assignée. Un type *union* est également défini, *ObjectState*, qui peut contenir soit une référence, soit une structure de type *PtrAndState.*

1. Dans la pratique, l'objet doit conserver une table qui recense les références et les éléments pointés. De cette manière on peut retrouver la référence à partir du pointeur et vice-versa.

Pointeur sur objet

`enum PtrKind {REF_ONLY,REF_AND_STATE};`
`struct PtrAndState {`
`  long ref;`
`  any  state;`
`};`
`union ObjectState switch(PtrKind) {`
`  case REF_ONLY: long ref;`
`  case REF_AND_STATE: PtrAndState ptr;`
`};`

Tableau 24 - Type IDL pour la désignation des pointeurs sur objet

Le langage IDL permet de définir des tableaux, le processus décrit ci-dessus s'applique donc de façon identique à tout tableau C++, quel que soit le type de ses éléments.

Une fois la structure IDL définie, le compilateur IDL est utilisé pour générer les fonctions d'empaquetage et de dépaquetage qui permettent de convertir, respectivement un état en *any* et un *any* en état.

IV.3.3.1.2 Sauvegarde

La sauvegarde de l'état correspond donc au fait de créer une structure contenant la valeur de chacun des attributs de l'objet et de renvoyer le *any* correspondant à l'empaquetage de cette structure. C'est le rôle de la méthode *Base_SaveState()*. La méthode *TranslateClass* de la métaclasse possède toutes les informations concernant les attributs de la classe d'entrée ainsi que celles concernant les éventuelles classes héritées. Elle peut donc générer le code de la méthode *Base_SaveState*, par le biais de la méthode *CreateSaveState*().

L'algorithme de *Base_SaveState*, relativement simple, est décrit ci-après.

- Créer la structure dans laquelle l'état sera stocké.
- Obtenir l'état correspondant aux éventuelles classes parentes par un appel à la méthode *Base_SaveState* de ces dernières, stocker ces états dans les champs prévus à cet effet dans la structure (récursion de l'obtention d'état dans une hiérarchie de classes ou d'objets).
- Remplir chaque champ de la structure correspondant à un attribut avec la valeur de cet attribut. Si cet attribut est de type pointeur sur objet, trois cas sont possibles :
  - l'objet désigné existe et n'a pas encore été sauvegardé : son état est obtenu par un appel à *Base_SaveState*, une référence lui est assignée. On stocke ces deux données dans une structure de type *PtrAndState* que l'on place dans une *union* de type *ObjectState*. Cette dernière *union* est empaquetée dans un *any* et stockée dans la structure.

  - l'objet désigné existe et a déjà été sauvegardé : sa référence est recherchée et stockée dans une *union ObjectState*. Cette union est empaquetée puis placée dans la structure sous la forme d'un *any.*
  - l'objet n'existe pas : la référence *null* lui est assignée puis stockée dans un *ObjectState*, celui-ci est empaqueté et placé dans la structure.
- Appeler la fonction d'empaquetage et retourner la variable de type *any* correspondante.

IV.3.3.1.3 Restauration

La restauration de l'état de l'objet est l'opération inverse de la sauvegarde, elle correspond donc à l'analyse de la structure d'entrée pour générer l'état de l'objet correspondant. Cette fonction est remplie par la méthode *Base_RestoreState(any state)* qui est également créée par la méthode *TranslateClass* de la métaclasse.

L'algorithme de *Base_RestoreState* est le suivant.

- Dépaqueter l'état fourni en entrée dans une structure.
- Pour chacun des champs qui correspond à l'état d'une classe parente, invoquer la méthode *Base_RestoreState* de cette classe.
- Pour chacun des champs qui correspond à un attribut de type pointeur sur objet, dépaqueter la valeur du *any* dans une *union* de type *ObjectState*. Ici encore, trois cas se présentent :
  - l'*ObjectState* contient une structure *PtrAndState* et la référence n'a pas déjà été restaurée : détruire l'objet pointé actuellement, créer un nouvel objet et faire pointer l'attribut sur celui-ci, invoquer la méthode *Base_RestoreState* avec la valeur de l'état contenue dans la structure et assigner à ce nouvel objet la référence également contenue dans la structure.
  - l'*ObjectState* ne contient pas une structure *PtrAndState* mais uniquement une référence et cette référence est différente de *null* : détruire l'objet précédemment pointé par l'attribut, assigner à l'attribut l'adresse de l'objet déjà créé et référencé.
  - la référence est *null* : détruire l'objet pointé par l'attribut et assigner à l'attribut la valeur *null*.
- Pour chacun des champs qui correspond donc à un attribut d'un autre type du langage, assigner à cet attribut la valeur contenue dans le champ de la structure.

IV.3.3.2 Etat partiel

Comme nous l'avons vu en III.1.3.2, il est parfois intéressant de ne sauvegarder que la partie de l'état de l'objet qui a été modifiée, partie que nous avons définie comme étant l'état partiel. Ce sous-ensemble de l'état complet est composé des attributs qui

ont été modifiés depuis la précédente sauvegarde. Ainsi on peut reconstruire l'état complet d'un objet à un instant donné à partir d'un état complet antérieur et des états partiels suivants, comme on le voit sur la figure 28.

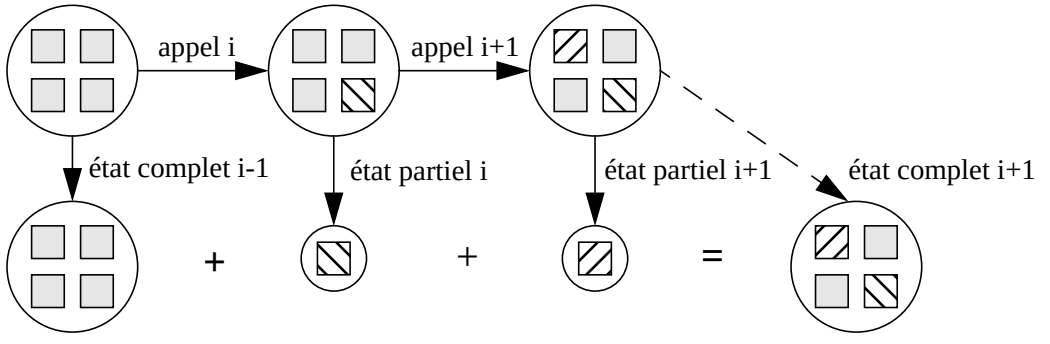

Figure 28 - Relation entre état complet et états partiels

Les deux méthodes du protocole à métaobjets qui permettent de construire un état partiel et de le restaurer sont, respectivement :

```
any Base_SavePartialState();
void Base_RestorePartialState(any partialState);
```

Ces deux méthodes étant optionnelles, elles ne font pas partie de l'interface *ReifiedObject* mais d'une autre interface, appelée *ReifiedObjectPS* qui hérite de *ReifiedObject* et qui définit simplement ces deux nouvelles méthodes.

Afin de construire l'état partiel de l'objet, il faut identifier les attributs qui ont été modifiés depuis la précédente sauvegarde de l'état. Pour ce faire la solution retenue est la suivante : nous utilisons OpenC++ pour générer les accesseurs définis en IV.2.1.3. Ainsi, à chaque fois qu'un attribut est modifié, un de ses accesseurs est appelé, ce dernier peut donc noter la modification de l'attribut. L'état partiel sera donc défini en fonction des attributs notés comme étant modifiés. Après chaque sauvegarde de l'état, tous les attributs retrouvent l'état non-modifié.

Dans les paragraphes suivants, nous présentons respectivement l'empaquetage et le dépaquetage de l'état partiel, la génération des accesseurs et enfin les techniques et algorithmes liés à la sauvegarde de l'état puis à la restauration.

IV.3.3.2.1 Empaquetage et dépaquetage

La structure d'un état partiel est par essence dynamique. Les attributs qu'elle contiendra ne seront connus qu'au moment de l'exécution. Il se peut, par exemple, qu'un état partiel soit vide si aucun des attributs n'a été modifié depuis la dernière sauvegarde, ou encore qu'il soit équivalent à l'état complet si tous les attributs de l'objet ont été modifiés.

Dans la pratique, on peut voir cette structure comme une séquence d'attributs. C'est comme cela que nous la représentons en IDL : une séquence de couples, chacun de ces couples contenant l'identifiant de l'attribut ainsi que sa valeur (cf. tableau 25).

En effet, il est nécessaire d'identifier chacun des attributs présents dans l'état partiel, puisqu'on ne connaît pas, à priori, la liste de ceux-ci. Le fait d'utiliser un *any* pour stocker la valeur de l'attribut, permet de définir la structure de façon générique.

```
typedef sequence<Attribute> PartialState;
struct Attribute {
   long id;
   any value;
};
```

Tableau 25 - Etat partiel en IDL

Cette représentation est sensiblement plus complexe lorsque l'attribut considéré est un tableau. En effet, dans un tableau, il est très courant que seuls quelques éléments soient modifiés. Dans l'optique de fournir un état partiel minimal, il est alors nécessaire d'identifier uniquement les éléments du tableau qui ont été modifiés. Chacun de ces éléments est identifié à partir de son index dans le tableau. L'état partiel d'un tableau est donc défini comme une séquence de couples contenant chacun l'index et la valeur de l'élément (cf. tableau 26). L'index d'un élément d'un tableau est constitué d'une séquence d'entiers. Effectivement, en C++, il est possible de définir des tableaux à multiples dimensions, l'index d'un élément est alors constitué de *n* entiers pour un tableau à *n* dimensions.

```
typedef sequence<ArrayElement> PartialArray;
struct ArrayElement {
   Index index;
   any value;
};
typedef sequence<long> Index;
```

Tableau 26 - Etat partiel d'un tableau en IDL

De cette manière, on peut sauvegarder l'état partiel d'un tableau dans une structure de type *PartialArray.* Cette structure peut être empaquetée et dépaquetée dans ou à partir d'un *any* grâce aux fonctions générées par le compilateur IDL. Elle peut donc être stockée dans une structure *Attribute*, laquelle, par le même mécanisme, peut prendre place dans la séquence *PartialState* qui correspondant à l'état partiel de l'objet.

A partir de ces définitions, on obtient une structure qui peut contenir un nombre variable d'attributs, la séquence *PartialState*. Les valeurs stockées dans cette séquence sont des couples qui correspondent à l'identifieur d'un attribut et sa valeur, la structure *Attribute*. La valeur de l'attribut est stockée sous la forme d'un *any.* De cette manière, l'attribut peut être de n'importe quel type : de base, complexe, tableau, objet, etc.

IV.3.3.2.2 Implémentation des accesseurs

Comme nous l'avons décrit précédemment, les accesseurs permettent de regrouper et de contrôler les différents accès aux attributs. Leurs avantages sont importants : d'une part ils permettent d'assurer une meilleure encapsulation, et d'autre part, ils simplifient l'identification de l'état partiel des objets. Nous présentons, dans cette partie, de quelle manière une métaclasse OpenC++ peut les implémenter.

OpenC++ réifie à la métaclasse les accès aux attributs, que ces accès soient en lecture, en écriture, ou par un opérateur post- ou pré-fixé (incrémentation, décrémentation, décalage, etc.). Ces réifications se font par le biais des méthodes *TranslateMemberRead*, *TranslateMemberWrite*, *TranslateUnaryOnMember* et *TranslatePostfixOnMember*. Notre métaclasse doit donc redéfinir ces différentes méthodes, afin de remplacer l'accès aux attributs par des appels à leurs accesseurs respectifs. La méthode principale *TranslateClass* peut alors créer trois accesseurs pour chacun des attributs : un accesseur de lecture, un accesseur d'écriture/lecture (préfixé) et un accesseur de lecture/écriture (postfixé). Ces accesseurs, outre de donner l'accès à l'attribut qu'ils contrôlent, permettent de lever le drapeau correspondant à l'attribut modifié.

Lorsque l'attribut en question est un tableau, les accesseurs correspondants ont besoin de connaître l'index de l'élément accédé. Leurs prototypes sont donc les suivants :

```
arrayType arrayRead( Index index);
arrayType arrayPostfix( Index index, arrayType value);
arrayType arrayPrefix( Index index, arrayType value);
```

Afin d'identifier chacun des éléments du tableau qui a été modifié, les deux accesseurs de lecture/écriture du tableau maintiennent une liste des indexes de ces éléments. Cette liste est alors vidée après chaque sauvegarde de l'état.

IV.3.3.2.3 Sauvegarde

La sauvegarde de l'état partiel repose donc sur la capacité d'identifier les attributs ayant été modifiés depuis la précédente sauvegarde. Grâce au système d'accesseurs présenté ci-dessus, les modifications d'attributs sont regroupées et faciles à identifier. Les accesseurs gèrent les drapeaux qui indiquent la modification de chacun des attributs. Ainsi, lorsqu'une sauvegarde partielle de l'état est demandée par le biais de la méthode *Base_SavePartialState*, il suffit de construire une liste de type *PartialState* qui contient chacun des attributs qui est identifié comme modifié. Le processus est alors quasiment identique à celui pour l'obtention de l'état complet :

- appel de la méthode *Base_SavePartialState* de chacune des classes mère et de chacun des objets-attributs[1] ; si l'état ainsi obtenu est non-vide, l'insérer dans la liste,

[1] La gestion des pointeurs sur objet est ici équivalente à la technique présentée en IV.3.3.1 pour l'obtention de l'état complet.

- test du drapeau de chacun des attributs de type simple ou complexe, si l'attribut a été modifié, créer une structure *Attribute*, la remplir avec l'identifieur et la valeur de l'attribut puis l'insérer dans la liste,
- pour les tableaux, construire une liste *PartialArray* à partir des éléments qui ont été modifiés (chacun d'entre eux est encapsulé dans un *ArrayElement* qui conserve l'index et la valeur),
- Remise à zéro de chacun des drapeaux et listes d'éléments de tableaux (on notera que lorsque l'option état partiel est activée, la méthode *Base_SaveState* doit également s'acquitter de cette tâche).

La méthode *Base_SavePartialState* est générée à partir de *TranslateClass*, comme c'est le cas de *Base_SaveState* pour l'état complet.

IV.3.3.2.4 Restauration

La restauration de l'état d'un objet à partir d'un état partiel se fait par un appel à la méthode *Base_RestorePartialState*. Cette méthode, également générée à partir de *TranslateClass*, est le pendant coté restauration de la méthode *Base_SavePartialState*. Elle a, par conséquent, un fonctionnement totalement symétrique et largement inspiré de *Base_RestoreState*.

IV.3.3.2.5 Etat partiel des objets Java

La technique présentée ci-dessus est totalement applicable à des objets Java. Il faut utiliser OpenJava pour générer les accesseurs spécifiques à Java que nous avons présentés en IV.2.2. A partir de ceux-ci et des drapeaux qu'ils maintiennent pour chacun des attributs, on peut générer des méthodes *Base_SaveState* et *Base_RestoreState* spécifiques qui fonctionnent comme les méthodes *Base_SavePartialState* et *Base_RestorePartialState* décrites ci-dessus. Pour rendre ce processus compatible avec la sérialisation, on fournira également des méthodes *writeObject* et *readObject* qui appellent *Base_SavePartialState* et *Base_RestorePartialState*.

L'unique particularité de Java, dans ce cas, concerne l'accès aux objets. En effet, comme nous l'avons vu précédemment, les objets Java sont toujours accédés par référence. Par conséquent, le mécanisme décrit pour les pointeurs sur objet en C++ doit être utilisé pour chacun des objets en Java.

Nous venons de traiter du protocole à métaobjet en ce qui concerne la partie serveur. La section suivante présente le pendant, côté client, du protocole.

IV.3.4 Génération des stubs

La génération du stub réifié qui correspond à un serveur est très dépendante de la compilation de ce même serveur. D'ailleurs, les interfaces *ReifiedStub* et *ReifiedObject* se ressemblent beaucoup. Plus précisément, le stub généré est fonction, d'une part, de la classe d'entrée (le serveur) et, d'autre part, de l'interface IDL du service

qu'implémente le serveur. En effet, un stub n'est qu'un mandataire, il offre donc la même interface (IDL) que le serveur qu'il représente. Dans notre contexte, il doit réifier les appels aux méthodes de cette interface à son métastub.

Les stubs standards de CORBA sont des classes relativement complexes. Outre la gestion des invocations, ils gèrent également un processus de création incluant la gestion de la référence, des méthodes d'empaquetage et de dépaquetage de la référence, etc. Plutôt que de devoir ré-écrire toutes ces méthodes, qui sont de plus relativement dépendantes de l'ORB utilisé, nous avons choisi de faire hériter notre stub réifié (*ReifiedStub*), du stub original créé par le compilateur IDL lors de la compilation de l'interface IDL du service. De cette manière, le stub réifié permet l'accès transparent à ces différentes méthodes et peut redéfinir les méthodes qu'il doit réifier : création, invocation du serveur et destruction.

En C++, l'unique particularité des stubs est qu'ils sont créés par une fonction statique (méthode de classe), qui appelle le véritable constructeur et initialise la référence. Cette méthode est la suivante :

```
static service_ptr _narrow(CORBA_Object_ptr reference_initiale);
```

Le processus général de création du stub réifié est le suivant. La métaclasse de la classe d'entrée connaît le nom de l'interface IDL par le biais d'un paramètre passé au compilateur OpenC++. Elle génère un nouveau fichier *"Sinterface.h"* qui contient la classe du nouveau stub. Cette classe hérite du stub original qui a le même nom que l'interface. Elle crée les méthodes *Base_StubCreate*, *Base_StubInvoc* et *Base_StubCleanup*, redéfinit les méthodes de l'interface, ainsi qu'un constructeur, un destructeur et la méthode statique *_narrow*. Cette dernière crée un objet puis appelle correctement le nouveau constructeur. Le constructeur initialise l'ORB, obtient la référence de la *MetaobjectFactory*, puis invoque sur celle-ci la méthode *MakeMetastub*. Enfin, elle appelle la méthode *Meta_StubCreate* de son métastub en lui fournissant la référence du serveur. De la même manière que les méthodes d'encapsulation (ainsi que *Base_Startup*, *Base_MethodCall* et *Base_Cleanup)* sont créés pour la classe du serveur, pour le stub on crée : *Base_StubCreate*, *Base_StubInvoc*, *Base_StubCleanup* ainsi que les méthodes d'encapsulation qui correspondent à l'interface IDL.

On notera que la méthode *Base_StubInvoc* qui sert à réaliser effectivement une invocation à partir du stub vers le serveur ne sera, en principe, pas utilisée dans notre protocole. En effet, le métastub envoie directement les invocations au métaobjet correspondant au serveur.

IV.3.5 Utilisation avec des librairies

L'application du processus que nous venons de décrire pour rendre réflexive une classe grâce à une métaclasse OpenC++ suppose d'avoir accès au code source des classes.

Nous avons néanmoins exposé en IV.2.1.3 une technique qui permettait de supporter les appels à des fonctions externes dont nous ne disposons pas du source (librairies «boîte-noire»). Le problème posé par ce type de fonction est que, étant donné que l'on ne dispose pas du code source de leur implémentation, on ne peut déterminer ce qu'elles font des arguments qu'on leur passe en paramètre. Lorsque l'on passe un attribut (ou un objet) en paramètre d'une telle fonction, on ne sait ni si cet attribut aura été modifié par cette fonction, ni si son adresse aura été conservée pour d'éventuelles modifications ultérieures, etc.

La solution que nous avons proposée passe par une utilisation préventive des accesseurs. Dans ce cas, on considère que l'attribut passé en paramètre sera toujours modifié, même si ce n'est pas le cas.

On notera que si la librairie utilisée propose des classes d'objets dont l'utilisateur hérite ou qu'il utilise directement, les objets issus de ces classes ne peuvent être utilisés en tant qu'attributs. En effet, notre technique ne permet de sauvegarder l'état que des attributs de type complètement connu. En revanche, de telles classes peuvent être instanciées dans des variables locales aux méthodes, ceci n'influant pas sur l´état des objets.

Il est envisageable de permettre à l'utilisateur d'exclure certains attributs du processus automatique de sauvegarde et de restauration de l'état. Mais dans un tel cas, qui permettrait l'utilisation de classes «boîte-noire» comme attributs, l'utilisateur devrait fournir lui-même des méthodes de sauvegarde et restauration pour chacun de ces attributs particuliers. Indépendamment de la méthode, ceci est une tâche très délicate, en particulier pour des librairies de classes donc «boîte-noire».

Un problème est néanmoins posé avec certaines structures allouées dynamiquement, comme les chaînes de caractères (char *) en C++. Une telle structure pointe sur un emplacement mémoire d'une taille donnée réservé à son usage. Elle peut utiliser tout ou partie de cette mémoire. Le problème est qu'il n'est pas aisé de connaître cette taille (à la compilation et même à l'exécution), seul le système d'exploitation pourrait la fournir. Lorsque l'on cherche à cloner une telle structure dans une variable temporaire, lors d'un appel à une fonction de librairie par exemple, on ne peut cloner que les données effectivement présentes dans la structure. C'est-à-dire qu'on ne peut pas créer un clone dont la taille allouée serait la même que la structure d'origine, on doit se contenter de la taille des données effectivement présentes dans la structure. Cela pose problème lorsque la fonction appelée écrit dans la structure des données d'une taille supérieure aux données d'origine.

Afin de résoudre ce problème, il faut rendre réflexive l'allocation mémoire, soit en utilisant un système d'exploitation réflexif, soit en fournissant ses propres fonctions d'allocation.

IV.4 Conclusion

Dans ce chapitre, nous avons proposé une implémentation du protocole à métaobjets défini au chapitre précédent. Cette implémentation est adaptée aux ORBs et langages du commerce car elle utilise la réflexivité à la compilation et la technique de l'encapsulation de méthode que nous avons présentée en III.2.1.1.2. L'usage de ces deux techniques permet d'ouvrir le système et les objets applicatifs, malgré l'utilisation de langages et d'ORBs «boîtes noires».

Outre la réalisation du protocole à métaobjets, la phase de compilation permet de contrôler l'usage du langage fait par le programmeur de l'application. Nous avons identifié plusieurs fonctionnalités offertes par le langage C++ qui sont incompatibles avec nos objectifs de sûreté de fonctionnement (visibilité des attributs, pointeurs, classes et fonctions amies et variables de classe ou globales). L'usage de la réflexivité à la compilation nous permet de filtrer l'utilisation de ces fonctionnalités et, dans la majeure partie des cas, d'automatiser des solutions de remplacement. C'est le cas des accesseurs : ces méthodes permettent d'assurer automatiquement une très bonne encapsulation des données. Elles définissent l'unique moyen d'accéder aux attributs d'un objet. Elles sont d'ailleurs utilisées ensuite pour l'identification de l'état partiel des objets.

Dans un deuxième temps, nous avons présenté les différentes techniques utilisées pour rendre réflexives les classes de l'application. Basées sur des mécanismes fournis par le compilateur OpenC++, ces techniques se veulent les plus transparentes possibles pour l'utilisateur. Malgré cet objectif de transparence, certains problèmes sont posés lors de l'usage de librairies «boîte-noire». En effet, nous ne disposons pas du code source de ces librairies, en conséquence de quoi il est difficile de leur faire confiance quant aux modifications qu'elles peuvent faire sur l'état des objets applicatifs. Nous proposons une solution qui consiste à sauvegarder préventivement l'état des objets avant l'appel à une fonction de librairie.

Nous avons donc montré, dans ce chapitre, la complémentarité qui existe entre réflexivité à la compilation, qui permet d'obtenir certaines informations structurelles non disponibles à l'exécution, et réflexivité à l'exécution qui est, par essence, plus dynamique. Le filtrage de conventions de programmation constitue un autre avantage de la réflexivité à la compilation. Le filtrage est, en effet, souvent nécessaire afin d'augmenter la confiance que l'on porte dans une application ou dans le support du langage qu'elle utilise.

L'approche présentée ici souffre toutefois de quelques limitations. En effet, puisque l'information destinée aux métaobjets est obtenue grâce à la réfléxivité à la compilation, c'est à dire au niveau du langage, les informations qui ne sont pas disponibles à partir de celui-ci ne peuvent être obtenues. C'est le cas par exemple des variables locales, qui font partie de l'état externe de l'objet et dont la sémantique réside dans les couches inférieures au langage (système d'exploitation ou ORB). Un autre problème, qui affecte certains mécanismes de réplication, concerne le déterminisme de l'application. Il est, en effet, impossible de déterminer exactement la

sémantique d'une application et donc d'identifier les sources possibles de non-déterminismes. Ces deux problèmes sont difficiles à identifier et à traiter. Pour les résoudre, une forme de réflexivité déclarative, qui implique une intervention de l'utilisateur, pourrait être utilisée comme c'est le cas dans ABCL/R2 [Masuhara *et al.* 1996]. Il s'agit alors de déclarer aux mécanismes les variables locales et les opérations nécessaires à leur gestion (sauvegarde et restauration). En ce qui concerne le déterminisme, il s'agit de déclarer au métaniveau les variables ou les évènements qui influent sur le déterminisme de l'application.

Les mécanismes utilisés ici pour implémenter le protocole à métaobjets ont été conçus pour pouvoir évoluer. En effet, lorsque les accesseurs CORBA (cf. I.2.2.3) seront complètement spécifiés et implémentés dans les ORBs du commerce, la phase de compilation réflexive ne sera plus nécessaire pour ce qui concerne le contrôle comportemental des objets. Nous avons également vu que lorsque le langage d'implémentation est capable de fournir lui-même l'état des objets, la technique que nous proposons s'y adapte et utilise les mécanismes fournis par celui-ci.

En définitive, la réflexivité à la compilation est très adaptée lorsque le système, l'ORB et le langage utilisés sont considérés comme des boites noires. De plus, comme nous l'avons vu, elle permet d'adapter le système lorsque ces couches inférieures offrent un certain niveau de réflexivité, ce qui permettrait de résoudre certaines des limites identifiées ci-dessus. Dans ce cas, le protocole à métaobjets reste le même et fournit donc une interface abstraite pour le contrôle des objets applicatifs. En effet, les spécifications du protocole décrit dans cette thèse sont génériques et restent valables quelle que soit la technique d'implémentation.

Les différentes propriétés de ce protocole à métaobjets sont illustrées à l'aide de l'architecture proposée au chapitre suivant.

CHAPITRE V

UNE ARCHITECTURE FLEXIBLE POUR LA SÛRETÉ DE FONCTIONNEMENT

L'objectif principal de ce chapitre est d'illustrer l'utilisation du protocole à métaobjets pour mettre en oeuvre des mécanismes de tolérance aux fautes dans un système CORBA. L'existence même du protocole à métaobjet permet de concevoir une architecture flexible. Nous proposons ici une architecture comprenant l'application, les métaobjets et des services pour la tolérance aux fautes. La pierre angulaire de l'architecture est constituée par le protocole à métaobjets, il fait le lien entre les différentes entités de l'architecture d'une manière souple et flexible tout en permettant une très bonne séparation entre application et mécanismes de tolérance aux fautes. Les services que nous proposons sont généraux, ils vont d'un service de communication de groupe à une fabrique de métaobjets, en passant par une fabrique d'objets qui permet de cloner les objets applicatifs sur différents sites du système. Cette architecture flexible peut, grâce au protocole à métaobjet, être adaptée aux besoins du système, aux nouvelles technologies ou encore à de nouveaux protocoles de communication.

Dans la première section, nous présentons l'architecture dans son ensemble : les services qu'elle propose, un exemple complet de métaniveau pour la tolérance aux fautes par réplication passive et quelques pistes pour la conception de nouveaux mécanismes de sûreté de fonctionnement. Nous abordons enfin brièvement le test de cette architecture.

Dans la seconde section nous présentons quelques résultats issus de l'utilisation du protocole à métaobjets au sein de cette architecture. Enfin nous concluons en mettant en exergue la flexibilité et l'ouverture fournie par la réflexivité et par l'architecture proposée.

V.1 Architecture

Dans l'architecture proposée, le protocole à métaobjets fait le lien entre objets et métaobjets d'une manière transparente et souple. Le but est ici d'illustrer les différentes propriétés de ce protocole à métaobjets. Cette architecture définit également les services nécessaires au bon fonctionnement de l'application et des mécanismes de tolérance aux fautes. Nous avons souhaité concevoir cette architecture de façon à ce qu'elle soit flexible et puisse s'adapter aux différents besoins du système en terme de mécanismes non-fonctionnels, de mécanismes réflexifs fournis par la plate-forme et de protocoles de communication de groupe disponibles sur celle-ci.

Dans cette section nous présentons tout d'abord l'architecture générale ainsi que les services qui la composent. Le protocole à métaobjets que nous avons conçu est ensuite illustré au sein de cette architecture par son utilisation pour implémenter des métaobjets de tolérance aux fautes par réplication passive. Nous proposons ensuite d'autres métaobjets de sûreté de fonctionnement, tels que des métaobjets d'injection de fautes et de test, qui illustrent la souplesse et la généralité du protocole à métaobjets. Enfin, nous jetons les bases de la validation de cette architecture et de son protocole à métaobjets.

V.1.1 Architecture générale et services

Nous présentons ici l'architecture proprement dite. La description de celle-ci est la plus générale possible mais les propos relatifs à son implémentation considèrent que le système d'exploitation utilisé est le système Unix.

L'architecture générale est illustrée par la figure 29 : elle est basée sur une plate-forme composée d'un système d'exploitation, d'un ORB, d'un protocole de communication de groupe et d'un ou plusieurs environnements d'exécution relatifs aux langages d'implémentation des objets. Les différents services, que nous allons définir ici, permettent d'abstraire les différents éléments de cette plate-forme : le protocole de communication de groupe est caché par le service de groupe et la notion d'unité d'exécution (processus, applications, etc.) propre au système est encapsulée par la fabrique d'objets. L'application repose sur les abstractions fournies par le modèle CORBA et sur l'environnement d'exécution propre au langage avec lequel elle est implémentée. Le protocole à métaobjets permet aux métaobjets d'effectuer le contrôle des objets applicatifs.

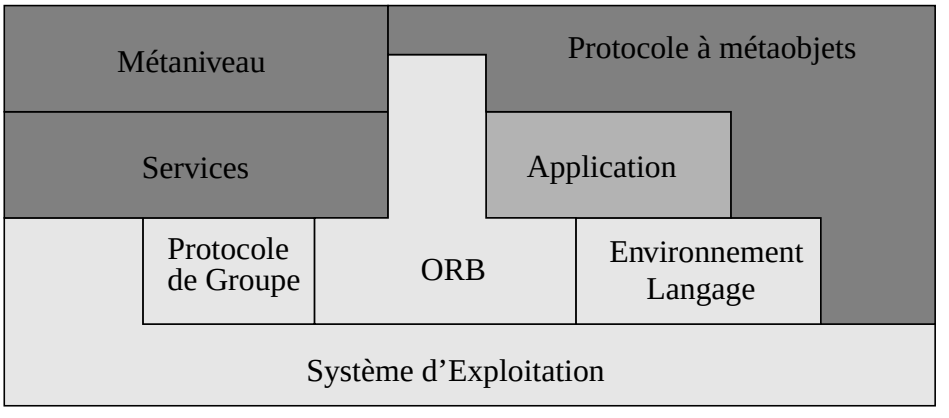

Figure 29 - Architecture Générale

Au chapitre précédent, nous avons identifié quelques-uns des intervenants de cette architecture :

- les serveurs, implémentés par les objets applicatifs, qui fournissent un service à leurs clients et qui utilisent le protocole à métaobjets,
- les métaobjets, au métaniveau, qui contrôlent le fonctionnement des serveurs suivant le protocole à métaobjets défini,
- les clients des serveurs, également au niveau applicatif, qui utilisent le service fourni par ces derniers et y accèdent à travers des stubs,
- les stubs, qui jouent le rôle de mandataires des serveurs dans l'espace d'adressage des clients et qui ont été rendus réflexifs (cf. IV.3.4),
- les métastubs, qui contrôlent les stubs et inter-agissent au métaniveau avec les métaobjets,
- la fabrique de métaobjets, au sein des services, qui crée les métaobjets et métastubs.

En plus de ces différentes entités, nous pouvons définir deux autres services, qui seront utiles dans l'architecture globale : une fabrique d'objets, qui permet de créer des répliques et un service de groupe, qui implémente, à la manière de l'OGS d'OpenDreams [Felber *et al.* 1996] (cf II.2.5), un protocole de communication de groupe utile à la réplication.

V.1.1.1 La fabrique d'objets

Dans le cadre de l'implémentation de mécanismes de réplication, nous avons besoin de pouvoir créer des répliques de serveurs sur différentes machines. Cette fonction est nécessaire lors de l'initialisation du système pour créer les répliques initiales ainsi qu'en cas de recouvrement pour créer de nouveaux exemplaires afin de ramener le nombre de répliques à son niveau original.

CORBA ne spécifie pas de correspondance entre objets et unités d'exécution mais dans la pratique, un objet est implémenté au sein d'une entité active (comme un processus dans les systèmes d'exploitation Unix ou une application sous Windows). De plus, dans notre contexte, pour des raisons de confinement d'erreur, il est préférable que chaque objet CORBA soit implémenté dans un processus Unix indépendant. Dans ce contexte, créer une réplique consiste à lancer un exécutable, ou, autrement dit, à créer un processus. La fabrique d'objet fait donc le lien entre objets CORBA et processus Unix. Présente sur chacun des sites du système, elle permet alors de créer des objets sur ces différentes machines.

La configuration de ce lien entre objets et exécutables se fait par un fichier de configuration, celui-ci contient une ligne par type d'objet susceptible d'être créé dynamiquement. Chaque ligne donne : le nom du service sous lequel les clients de la fabrique d'objets demanderont sa création, le chemin d'accès complet et le nom de l'exécutable correspondant à l'implémentation de ce service ainsi que les arguments nécessaires à celui-ci (à la manière de /etc/inetd.conf pour les services Unix). Certains des arguments de l'exécutable sont définis statiquement dans ce fichier de configuration, et d'autres, sont donnés dynamiquement par le client de la fabrique. Un exemple de fichier de configuration est donné dans le tableau 27. Dans cet exemple trois services sont définis, le chemin d'accès de chacun des exécutables correspondant est donné ainsi que ses paramètres. Le *ServiceX*, par exemple, prend quatre paramètres : le premier et le troisième sont statiques alors que le second et le dernier sont dynamiques.

Exemple de fichier de configuration de la fabrique d'objet

`ServiceX: /opt/Banque/Securite/ServeurDeClefs -optimize $1 -GTE $2`
`ServiceY: /opt/Banque/Fichier/ServeurDeFichiers`
`GroupSrv: /opt/Groupe/GrpDaemon -config=/etc/groupe.cnf $1`

Tableau 27 - Fichier de configuration de la fabrique d'objets - exemple

La fabrique d'objets fournit deux méthodes pour la création d'un objet. La première se contente de créer le processus correspondant à l'objet alors que la seconde, une fois l'objet créé, renvoie à son client la référence de celui-ci. Cette dernière méthode implique un protocole entre l'objet créé et la fabrique de métaobjet afin que cette dernière puisse récupérer la référence. Ce protocole (illustré figure 30) est le suivant :

1- le client du service invoque la méthode *launch_with_ref* pour demander la création d'un objet,

2- la fabrique d'objets effectue les appels systèmes nécessaires à la création de l'objet au sein d'une unité d'exécution indépendante,

3- l'objet créé envoie sa référence à la fabrique d'objets à travers la méthode *callback_ref*,

4- la fabrique d'objet retourne la référence de l'objet qu'elle vient d'obtenir à son client.

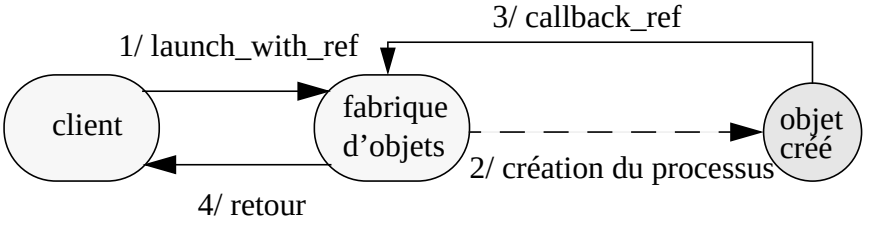

Figure 30 - Processus de création d'un objet et de récupération de sa référence

Finalement, l'interface de la fabrique d'objets est donnée tableau 28.

Fabrique d'Objets

`interface ObjectFactory {`
`  void launch(in string ObjectName, in string[] args);`
`  object launch_with_ref(in string ObjectName, in string[] args);`
`  void callback_reference(in object reference);`
`};`

Tableau 28 - Interface IDL de la fabrique d'objets

V.1.1.2 La fabrique de métaobjets

Tel que nous l'avons décrit dans le chapitre précédent, le rôle de la fabrique de métaobjets est de créer un métaobjet (ou métastub) pour chaque objet (stub réifié) de l'application. En effet, le protocole à métaobjets spécifie que chaque objet, dans son constructeur, demande à la fabrique de métaobjets de lui créer un métaobjet. Cette demande se fait par le biais de la méthode *MakeMetaobject*. Lors de sa destruction, l'objet demande également celle de son métaobjet via la méthode *DestroyMetaobject*. La fabrique de métaobjets est donc garante du cycle de vie des métaobjets. Elle peut, lorsque l'utilisateur le souhaite ou que la stratégie de tolérance aux fautes le requiert, changer dynamiquement le métaobjet associé à un objet, par exemple pour changer le mécanisme de réplication utilisé. L'interface du service est donnée au tableau 29. La méthode *findMO* permet d'obtenir la référence du métaobjet associé à l'objet donné en paramètre.

Interface IDL de la fabrique de Métaobjets

`interface MetaobjectFactory {`
`  Metaobject MakeMetaobject(in string class, in ReifiedObject o);`
`  void DestroyMetaobject(in Metaobject mo);`
`  Metastub MakeMetastub(in string class, in ReifiedStub s);`
`  void DestroyMetastub(in Metastub ms);`
`  Metaobject findMO(in Object o);`
`};`

Tableau 29 - Interface IDL de la fabrique de métaobjets

Au delà de cette gestion du lien entre objets et métaobjets et du cycle de vie de ces derniers, la fabrique de métaobjets est également responsable du choix du type de métaobjet qu'elle crée pour les différents objets, autrement dit, de la classe des métaobjets. On peut en effet imaginer que différents objets de l'application qui prennent place dans la même architecture n'utilisent pas tous la même technique de réplication, les uns peuvent utiliser la réplication active alors que les autres utilisent la réplication passive. Chacun de ces mécanismes peut être implémenté dans une classe de métaobjets différente. La fabrique de métaobjets doit donc sélectionner le type de métaobjet en fonction de la stratégie définie par le concepteur du système.

Exemple de fichier de configuration de la fabrique de métaobjet

`ServiceX: ActiveReplicationMO arg1 arg2`
`ServiceY: PassiveReplicationMO arg1`
`ServiceZ: DebugMO`
`ServiceM: FaultInjectorMO 10000 -mode=bit_flip`

Tableau 30 - Fichier de configuration de la fabrique de métaobjets - exemple

Nous proposons de décrire la stratégie de tolérance aux fautes, et plus généralement les couples associant classe d'objet et classe de métaobjet, dans un fichier de configuration de la fabrique de métaobjet. A la manière du fichier de configuration de la fabrique d'objets, ce fichier devra donner sur chaque ligne : la classe de l'objet, la classe du métaobjet correspondant et d'éventuels arguments. Un exemple d'un tel fichier est donné dans le tableau 30. Les informations relatives à la tolérance aux fautes seule, comme la description des domaines de réplication, sont du ressort des métaobjets de tolérance aux fautes. Ils définiront donc leurs propres fichiers de configuration dans cette optique.

A partir de cette spécification, on peut faire deux choix d'implémentation différents : les métaobjets peuvent être crées directement au sein de la fabrique de métaobjets ou, au contraire, il peut être utile de les créer dans un processus distinct pour des raisons de performances. Dans ce dernier cas, la fabrique de métaobjets utilise les services de la fabrique d'objets pour créer chaque métaobjet dans un processus indépendant. Le protocole incluant objets, métaobjets, fabrique de métaobjets et fabrique d'objets est illustré figure 31. Les objets demandent la création des métaobjets à la fabrique de métaobjets, celle-ci utilise la fabrique d'objet pour effectivement créer le métaobjet, fabrique d'objets et métaobjets suivent le protocole précédemment défini. D'autre part, lors de l'initialisation du système ou encore lors d'un recouvrement, les métaobjets peuvent avoir besoin de créer de nouvelles répliques d'un objet applicatif. Pour ce faire, ils peuvent utiliser les service de la fabrique d'objets.

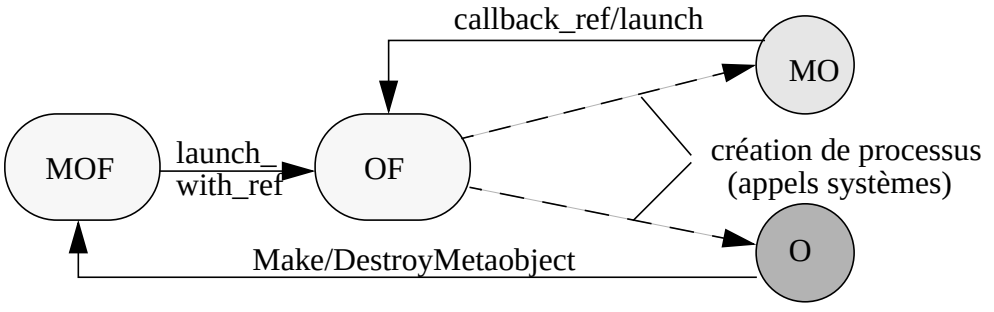

Figure 31 - Protocole entre Objets, Métaobjets et Fabriques

Parallèlement à la fabrique d'objets, la fabrique de métaobjets est présente sur chacune des machines du système sur laquelle un ou plusieurs métaobjets sont susceptibles d'être créés. Ainsi, la défaillance d'un site ne pénalise pas d'autre site du point de vue de ces deux services.

V.1.1.3 Service de groupe

Dans un souci de séparation des responsabilités, il nous semble très bénéfique d'avoir un service pour la gestion des protocoles de groupe, à la manière de l'OGS dans OpenDreams (cf. II.2.5), dans notre architecture. En effet, les métaobjets de tolérance aux fautes utilisent de tels protocoles pour communiquer avec l'ensemble des répliques d'un même objet. Un protocole de groupe permet non seulement d'abstraire la notion de groupe, ce qui permet d'envoyer des messages à plusieurs entités comme on le ferait à destination d'un simple objet, mais il fournit plusieurs propriétés très utiles aux mécanismes de tolérance aux fautes. La plus importante de ces propriétés est l'atomicité des messages : si un message est reçu par un des membre du groupe, alors tous les membres non-défaillants le reçoivent. Et une seconde propriété simplificatrice est celle de l'ordre des messages : chacun des membres non-défaillant du groupe reçoit les mêmes messages dans le même ordre. Par conséquent, l'utilisation d'un service de groupe permet, non seulement, de rendre plus indépendants les métaobjets du protocole de groupe effectivement utilisé mais également de rendre les mécanismes de tolérance aux fautes moins complexes.

Le service de groupe que nous avons réalisé utilise de façon interne le protocole xAMp [Veríssimo et Marques 1990] mais implémente une interface générique qui permet d'envisager d'utiliser un protocole différent. Il permet à des objets CORBA de créer un groupe, de s'inscrire dans un groupe, d'envoyer et de recevoir des messages destinés à un groupe, d'obtenir la liste des membres d'un groupe et enfin, d'etre averti lors d'une modification de la liste des membres du groupe (ceci sera utilisé notamment pour la détection de défaillance).

Les différentes entités qui composent le service (cf. figure 32) sont :

- Les *GroupAdmin (GA)*, qui sont responsables de la gestion d'un groupe sur un site,
- Les *GroupMember (GM)*, qui sont responsables des relations entre un membre et son groupe, ils réalisent l'interface entre les membres et le service de groupe. En tant qu'interface, ils permettent de changer aisément l'implémentation du service de groupe sans conséquence pour celle des membres,
- La *GroupAdminFactory (GAF)*, responsable de la création des *GroupAdmin* sur chaque site.

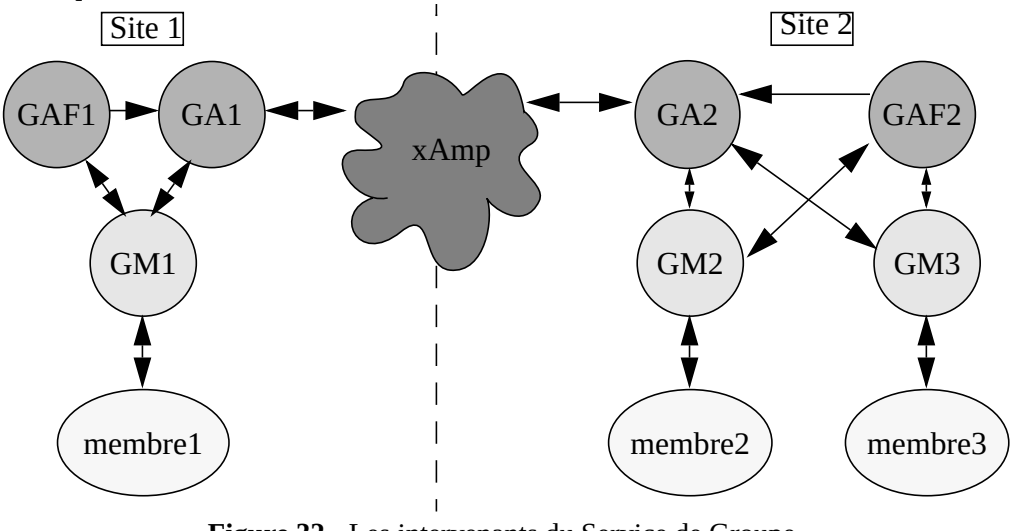

Figure 32 - Les intervenants du Service de Groupe

Une description du protocole de groupe implémenté déborderait du cadre de cette thèse, nous nous contentons de donner et de décrire l'interface *GroupMember* (tableau 31), qui permet à un objet d'utiliser le service simplement.

GroupMember

```
interface GroupMember {
  void open();
  void close();
  void join();
  void leave();
  void broadcast(in Any message);
  oneway void broadway(in Any message);
  void change_view( in view newView);
  void deliver(in Any message);
};
```

Tableau 31 - Interface IDL GroupMember du Service de Groupe

Cette interface est classique, elle permet d'ouvrir l'accès à un groupe existant (méthode *open*), si le groupe n'existe pas il sera alors créé. La méthode *join* permet de s'inscrire dans un groupe précédemment ouvert alors que *leave* permet de le

quitter. Le groupe peut être fermé grâce à la méthode *close*, dans ce cas, le groupe est automatiquement quitté. Enfin deux méthodes permettent d'émettre des messages à destination du groupe : *broadcast*, qui est bloquante, et dont le retour signifie que chacun des membres du groupe a reçu le message, et *broadway*, qui est non bloquante, et avec laquelle l'émetteur saura que les membres du groupes ont reçu le message lorsque il recevra lui-même le message émis. En ce qui concerne les méthodes d'entrée, *change_view* est appelée lorsque qu'un changement de vue survient, c'est-à-dire lorsqu'un membre s'inscrit ou quitte le groupe, et *deliver* permet de recevoir les messages émis à destination du groupe.

On notera que les méthodes *change_view* et *deliver* ne font que passer l'information du *GroupAdmin* au membre du groupe par simple appel de méthode (notion d'*upcall*). La méthode *change_view* pourra être utilisée pour la détection de défaillance d'un site distant et ainsi déclencher le processus de recouvrement.

V.1.1.4 Intérêt de métaniveaux multiples

Un métaobjet étant un objet, il est possible de lui appliquer également le protocole à métaobjets afin de le rendre lui-même réflexif. Le métaobjet est donc, dans ce cas contrôlé par un méta-métaobjet. Cette technique a déjà été utilisée dans FRIENDS [Fabre et Pérennou 1998] pour composer les mécanismes non-fonctionnels. Nous l'utiliserons également ici pour les mêmes raisons. La figure 33 montre un exemple où sont représentés un client et un serveur avec deux métaniveaux.

Dans cet exemple, le serveur possède un métaobjet de tolérance aux fautes (*MO1*), qui implémente un protocole de réplication passive, par exemple. Ce métaobjet possède lui même un métaobjet (*MO2*, le méta-métaobjet du serveur), qui gère un protocole de communication fiable. Le client accède au serveur à travers un stub réifié (*S_S*). Le métastub (*MS1*) de ce stub est client du métaobjet de tolérance aux fautes. Il accède à ce métaobjet à travers un stub (*S_MO1*), celui-ci est également réifié et son métastub (*MS2*) est client du métaobjet de communication, il communique avec lui suivant un protocole de communication fiable. Etant donné que le métaobjet de communication constitue le métaniveau le plus haut (il n'est pas réifié), le métastub de dernier niveau y accède à travers un stub standard (*S_MO2*), c'est-à-dire non-réifié.

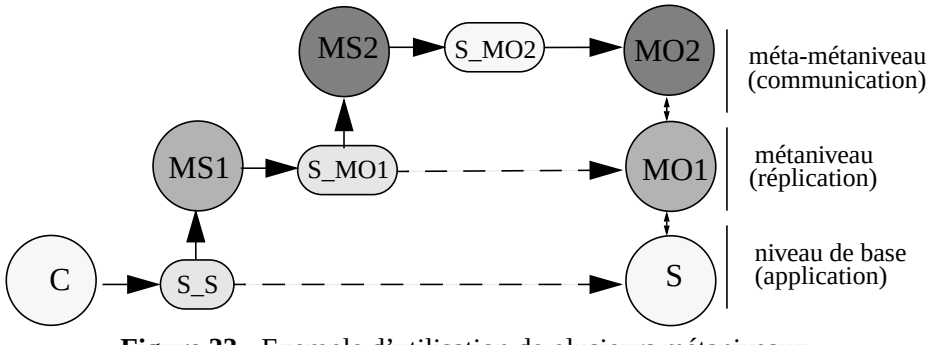

Figure 33 - Exemple d'utilisation de plusieurs métaniveaux

Cet exemple illustre la composition de deux propriétés non-fonctionnelles, la tolérance aux fautes et la communication fiable. Chacun des métaobjets et métastubs gère sa propre propriété, et celle-ci est transparente pour les autres couches. Comme nous le verrons dans l'architecture générale, et particulièrement pour la conception des métaobjets de tolérance aux fautes, cette séparation est des plus pratiques. Nous utiliserons notamment ceci pour gérer le protocole de groupe à un niveau supérieur à la tolérance aux fautes. Ainsi les métaobjets de tolérance aux fautes n'auront pas à se préoccuper de la gestion des groupes.

V.1.2 Métaobjets de tolérance aux fautes

La partie centrale de cette architecture est, bien évidemment, la couche de tolérance aux fautes. Nous avons choisi d'illustrer le protocole à métaobjets et l'architecture décrite ci-dessus avec un mécanisme de réplication passive simplifié. Nous décrivons ici sa conception et son implémentation.

La description de haut-niveau d'un algorithme de réplication passive peut sembler, au premier abord, simple. Pourtant cet algorithme est, dans la pratique, relativement complexe, notamment en ce qui concerne le traitement de la reconfiguration suite à une défaillance d'une des répliques. Pour la maîtrise de la conception de cet algorithme, nous avons choisi de le modéliser avec des Statecharts[1], ce qui permet également de simuler son fonctionnement. Cette simulation s'est révélée très fructueuse puisque de nombreuses fautes de conception ont été découvertes et corrigées lors de cette phase de la conception.

A un haut-niveau d'abstraction, quatre types d'évènements principaux sont à traiter par un tel algorithme :

- L'invocation d'une méthode du serveur,
- Le recouvrement suite à une défaillance d'une des répliques,
- La réception d'une copie de l'état du primaire,
- Le clonage d'une nouvelle réplique.

Le traitement à effectuer suite à la réception d'un tel évènement varie en fonction de l'état actuel de la réplique considérée, si la réplique est primaire ou secondaire, si un recouvrement est en cours, etc.

Le but de cette section n'étant pas de détailler le mécanisme lui-même mais plutôt d'illustrer l'utilisation du protocole à métaobjet, nous ne donnerons qu'une vue relativement générale du mécanisme.

La figure 34 représente la modélisation abstraite de ce protocole en utilisant les Statecharts. Deux états représentent les fonctionnements en mode primaire ou secondaire. Le fonctionnement mode primaire est relativement plus simple : lors du fonctionnement normal (en l'absence de faute), le métaobjet reçoit et répond à des invocations (*invoke*) ; lorsqu'une défaillance survient (*recover*), il clone une nou-

1. Les Statecharts [Harel 1996] sont des automates à état finis, hiérarchisés, concurrents et semi-formels.

velle réplique. En revanche, le mode de fonctionnement secondaire implique de stocker les invocations, de traiter les copies d'état reçues du primaire (*checkpoint*), de mettre à jour son état lors de la phase de clonage (*cloning*) et éventuellement d'activer une phase de recouvrement (*recover*) lors d'une défaillance du primaire.

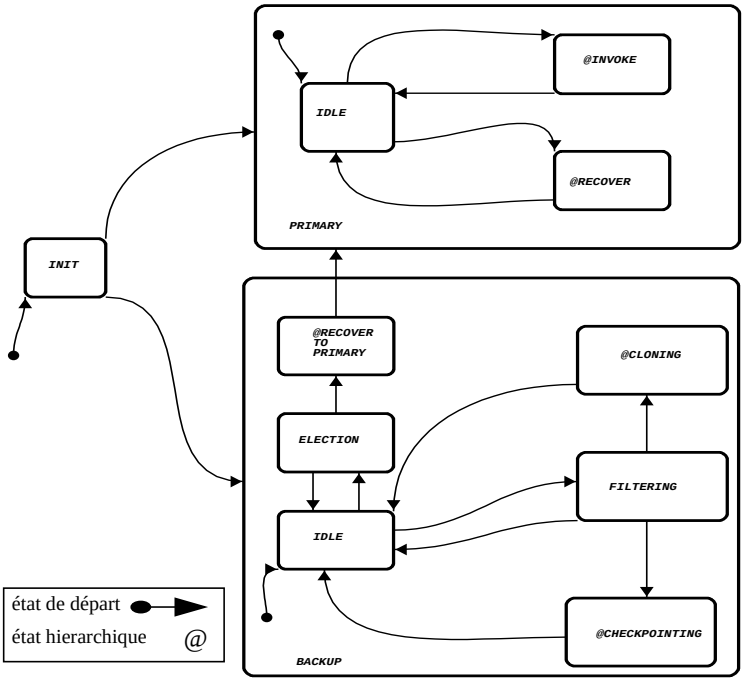

Figure 34 - Modélisation Statecharts du métaobjet de réplication passive

V.1.2.1 Traitement des invocations

Le traitement des invocations représente la majeure partie du traitement demandé à un serveur puisqu'il correspond au fonctionnement en l'absence de fautes. Dans le cadre de réplication passive, seul le primaire réalise effectivement les invocations. Les secondaires se contentent de mettre à jour leur état en fonction des informations reçues du primaire, ainsi que de stocker les invocations dans le cas d'une défaillance du primaire.

A chacune des invocations reçues, un identificateur unique est associé ; on considère que pour chacune des répliques, l'ordre de réception est le même, propriété ici fournie par le service de groupe. Chacune des répliques associe donc le même identificateur à chacune des invocations qu'elle reçoit. Côté secondaire, l'identificateur est stocké avec l'invocation et coté primaire, l'identificateur est envoyé avec l'état qui fait suite à l'invocation. Autrement dit, après avoir réalisé une invocation, le primaire obtient l'état du serveur et l'envoie aux répliques accompagné de la réponse faite au client et de l'identificateur de l'invocation. Ainsi, les secondaires peuvent purger l'invocation de leur file d'attente puisqu'elle a été traitée par le primaire. Ils conservent néanmoins la réponse faite par ce dernier au client, car, en cas de

défaillance du primaire, elle sera nécessaire[1] afin de ne pas répéter une exécution de méthode déjà effectuée. Ils appliquent l'état reçu du primaire à leur objet. Lorsqu'il répond au client, le primaire renvoie également cet identificateur, ainsi le client pourra filtrer les doublons éventuels.

V.1.2.2 Défaillance d'un secondaire

Lorsqu'une réplique défaille, les autres répliques en sont averties par le biais d'un changement de vue du groupe auquel elles appartiennent car, dans ce cas, le service de groupe averti chacun des membres qu'un membre a quitté le groupe. Lorsque c'est une réplique secondaire qui défaille, le primaire doit reconfigurer le système pour qu'il retrouve un nombre de répliques correct. Pour ce faire, il demande à la fabrique d'objets de créer une nouvelle réplique sur le site sélectionné. Il envoie à cette nouvelle réplique une copie de son état accompagnée de l'identificateur de la prochaine invocation.

Lorsque sa création est terminée, la nouvelle réplique peut recevoir des invocations issues des clients avant de recevoir le message contenant l'état du primaire. Elle ne peut par contre assigner d'identificateur correct à ces invocations puisqu'elle ne connaît pas le passé du groupe de répliques, elle se contente donc de stocker ces invocations avec un identificateur temporaire. Lorsqu'elle reçoit l'état complet du primaire, elle peut re-calculer les identificateurs corrects des requêtes mises en attente à partir de l'information fournie par le primaire. Elle retrouve alors le statut de secondaire correctement initialisé.

V.1.2.3 Défaillance du primaire

Une défaillance du primaire est bien plus complexe à gérer que celle d'un simple secondaire. En effet, le processus qui mène un secondaire à se reconfigurer en primaire implique de nombreuses activités de reconfiguration.

Suite à la défaillance du primaire, les secondaires doivent tout d'abord élire un nouveau primaire. Ce dernier doit créer un nouveau secondaire qui le remplacera et lui envoyer son état pour qu'il se reconfigure correctement. Afin de prévenir toute famine d'un client, le nouveau primaire ré-émet la réponse à la dernière invocation traitée ; il avait stocké cette réponse lorsqu'il a reçu le dernier état issu du précédent primaire. Cette ré-émission est nécessaire pour traiter le cas où la réponse n'aurait pas été correctement envoyée au client avant la défaillance du primaire. Ensuite, il traite toutes les invocations de sa file d'attente, celles-ci n'ont en effet pas été traitées par le primaire ayant défailli. En effet, dans la pratique, les requêtes reçues en tant que secondaire sont associées à des informations complémentaires ; le nouveau primaire doit donc traiter tout d'abord ces requêtes avant de ré-initialiser ses files d'attente et de jouer pleinement le rôle de primaire.

1. Comme nous nous le présentons en V.2.1.3, en cas de défaillance du primaire, la précédente réponse à une invocation est systématiquement ré-émise.

La figure 35 illustre ce processus. Tout d'abord le nouveau primaire obtient l'état de son objet de base (état *Local_SaveState*), puis il crée une nouvelle réplique secondaire (via la fabrique d'objets) à laquelle il envoie son état (état *Remote_Clone*). La boucle qui commence à partir de l'état *Handle_Pending_Queue* correspond à la gestion des invocations en attente. Pour chacune des requêtes de sa file d'attente, le nouveau primaire effectue l'invocation sur son objet de base (*Local_Invoke*), copie son état (*Local_SaveState_2*) et envoie cet état à ses répliques secondaires (*Remote_Checkpoint*). Une seconde boucle commence à partir de l'état *Flush_Queue*, son but est de vider la queue des messages qui ne seront plus nécessaires. En effet, les secondaires utilisent certains messages, tels que les copies d'état du primaire, qui sont inutiles au primaire. Une fois la file d'attente vidée, la procédure de recouvrement est terminée et le primaire retrouve un fonctionnement normal.

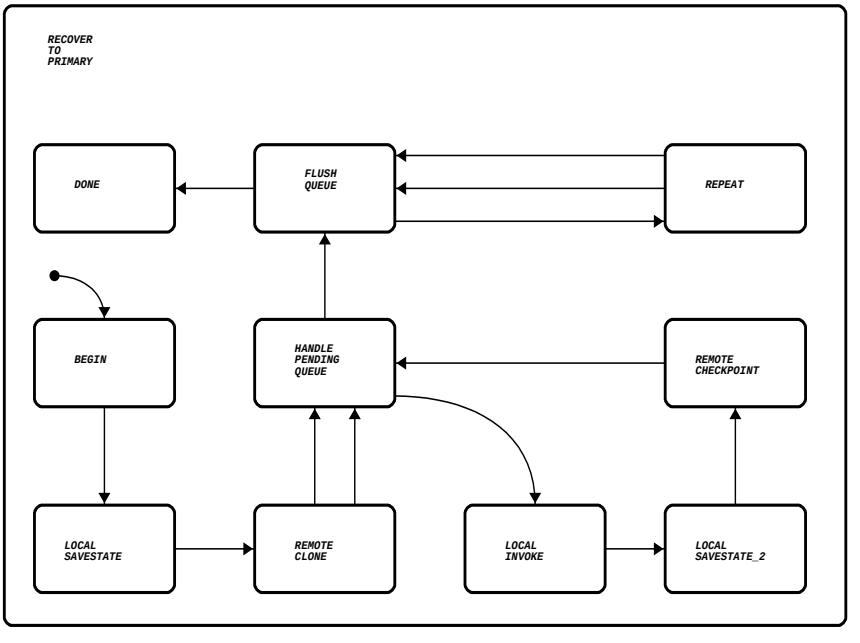

Figure 35 - Statechart-Recouvrement par un secondaire

V.1.2.4 Adéquation du protocole à métaobjets

On se rend bien compte sur cet exemple de réplication passive, que la mise en œuvre de mécanismes de réplication est une activité indépendante de l'application. A aucun moment de son cycle le mécanisme n'a besoin de savoir ce que réalise l'application, il suffit de pouvoir activer ses méthodes, obtenir ou restaurer son état, savoir la cloner, etc. Même si l'algorithme de ce mécanisme peut paraître complexe, il n'en est rien si on le compare à un même mécanisme intégré à l'application. La séparation des concepts permise par le protocole à métaobjets entre l'objet qui représente la fonction de l'application et le métaobjet, qui représente les propriétés non-fonctionnelles, est grandement simplificatrice. Toutes les informations néces-

saires à la réplication d'objets CORBA (qui sont disponibles sur une plate-forme COTS) sont fournies au métaobjet. Ce dernier dispose également des moyens d'activation nécessaires.

Dans l'exemple de mécanisme de tolérance aux fautes que nous avons présenté, nous n'avons pas traité le problème des invocations imbriquées qui se présente lorsqu'un serveur répliqué *S1* est client d'un autre serveur répliqué *S2*. Dans ce cas, si la réplique primaire de *S1* défaille juste après avoir invoqué une méthode de *S2*, le primaire *S1* qui la remplace ne doit pas invoquer une seconde fois la méthode de *S2* sous peine de modifier son état de façon incohérente. Ce problème est difficile à résoudre car l'invocation de *S2* par *S1* se fait en plein milieu d'une méthode de ce dernier, et qu'il est difficile de contrôler les actions internes à une méthode sans utiliser un support d'exécution réflexif. Dans notre cas, on pourra utiliser un protocole spécifique entre métastubs et métaobjets pour éviter d'effectuer une invocation identique plusieurs fois tout en permettant d'obtenir à chaque fois la réponse correcte.

Finalement, et malgré ses limites dues à l'opacité de la plate-forme sous-jacente, ce protocole à métaobjets convient parfaitement à l'implémentation de mécanismes de tolérance aux fautes mais également, comme développé en V.1.3, permet bien d'autres utilisations liées à la sûreté de fonctionnement de manière plus générale.

V.1.2.5 Simulation et autres avantages de la modélisation

L'outil que nous avons utilisé pour la modélisation Statechart de ce mécanisme, Statemate®, permet la simulation des automates décrits. On peut grâce à cet outil, faire avancer la simulation pas-à-pas, transition par transition, observer les structures de données, files d'attente de messages entre autres. Ce processus de simulation nous a permis de valider, dans une certaine mesure, l'algorithme du mécanisme à travers son modèle. Même si cette simulation n'aide pas au test de l'implémentation correspondante, elle nous a permis d'augmenter la confiance que nous portons dans sa conception.

La simulation a, par exemple, mis en lumière certains dysfonctionnements de la gestion de la file des invocations lors d'un recouvrement de défaillance du primaire. Identifier cette faute de conception lors de l'implémentation aurait été bien plus difficile du fait de la complexité des structures de données et du parallélisme du système.

La modélisation par Statecharts ne prend pas en compte la notion d'objets ou de métaobjets, elle s'attache au caractère fonctionnel du système représenté. Néanmoins, la communication par messages et par files d'attente facilite la modélisation des inter-actions entre objets et métaobjets. Cette description fonctionnelle peut, par la suite, être utilisée pour la conception UML de l'application. Ceci est d'autant plus vrai que les états de haut niveau de la figure 34 représentent les fonctions générales du mécanisme et correspondent dans l'implémentation aux méthodes du métaobjet.

V.1.2.6 Intégration des métaobjets dans l'architecture

Comme nous l'avons vu précédemment, le métaobjet de tolérance aux fautes prend place au dessus du serveur dans le premier métaniveau. Au-dessus de celui-ci se trouve un métaobjet de communication de groupe qui reçoit les messages du service de groupe et les transforme en invocation pour le métaobjet de tolérance aux fautes. Certaines de ces invocations n'engendrent pas de réponse, c'est le cas par exemple du métaobjet de réplication passive en mode secondaire qui ne traite pas les requêtes mais se contente de les stocker. Le métaobjet de groupe ne sait a fortiori pas dans quel mode, primaire ou secondaire, son objet de base (le métaobjet de réplication passive) se trouve puisqu'il est indépendant de son caractère fonctionnel. Le mode de fonctionnement, le fait de répondre ou de ne pas répondre, sont des décisions du ressort du métaobjet de tolérance aux fautes et dans un souci de séparation des responsabilités, il faut préserver cette indépendance. C'est pourquoi, pour répondre aux invocations du client, le métaobjet de tolérance aux fautes utilise un stub (qui sera réifié pour utiliser le service de groupe) et communique directement avec le client. Le métaobjet de tolérance aux fautes prend lui-même l'initiative de répondre ou de ne pas répondre et cette décision n'influe pas sur le métaobjet de communication de groupe.

L'architecture de métaniveau correspondant à ce choix de conception est illustrée figure 36. Dans cette architecture, le chemin pris par les invocations et leurs réponses peut paraître surprenant. En effet, l'invocation arrive au métaobjet de tolérance aux fautes par son propre métaobjet mais la réponse ne passe pas par celui-ci. La justification de ce choix de communication dissymétrique (la réponse ne suit pas le chemin de la requête) réside dans la logique de séparation des responsabilités des métaobjets. Une approche symétrique aurait conduit les deux métaobjets, de tolérance aux fautes et de groupe, à communiquer ou à partager de l'information afin de déterminer s'il fallait envoyer une réponse au client ou pas, réduisant à néant leur indépendance mutuelle. On notera également que le métaobjet de tolérance aux fautes accède au groupe de répliques à travers un stub réifié. En effet, on peut dire que le primaire est le client des secondaires : il invoque certaines de leurs méthodes, pour leur fournir son état notamment. Le stub *S_MFT* permet donc d'accéder à ceux-ci ; il est contrôlé par un métastub de gestion du protocole de groupe qui permet, encore une fois, de rendre plus indépendants les différents métaniveaux.

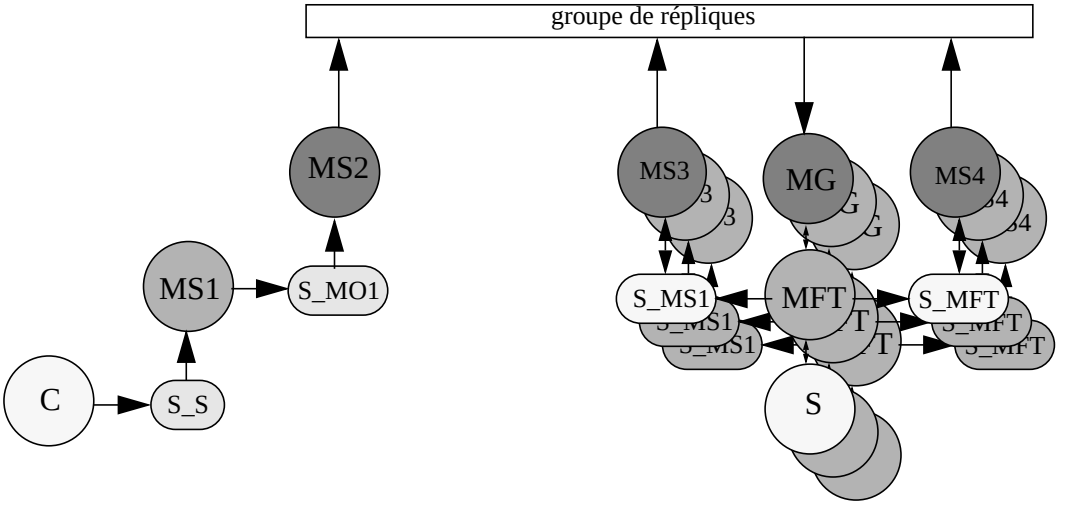

Figure 36 - Architecture complète du métaniveau

V.1.3 Autres métaobjets de sûreté de fonctionnement

Le protocole à métaobjets que nous proposons, ainsi que l'architecture CORBA qui le met en œuvre, permet d'implémenter bien d'autres mécanismes que l'exemple de réplication passive décrit ci-dessus. En effet, les mécanismes de réplication semi-active ou encore active ne nécessitent pas plus d'information que celui-ci, ils peuvent donc être facilement implémentés au sein de métaobjets sur cette architecture. De plus, grâce à une conception orientée-objet, il est aisé d'identifier de nombreux points communs à ces différents mécanismes. Ainsi, des parties substantielles de conception et d'implémentation peuvent être réutilisées lors du développement de nouveaux mécanismes. Eventuellement, certains automates développés pour la réplication passive pourront être ré-utilisés.

Au delà des mécanismes de tolérance aux fautes par réplication, nous pensons que ce protocole à métaobjets peut être utilisé dans d'autres domaines de la sûreté de fonctionnement. Nous présentons ici quelques pistes concernant le test et l'injection de fautes.

V.1.3.1 Métaobjets de test

Nous pensons que le protocole à métaobjets que nous avons conçu en premier lieu pour la tolérance aux fautes peut être utile au test des applications. En effet, la technique du test logiciel consiste d'une part à concevoir des jeux de tests (suite de méthodes -et paramètres- à appliquer à un objet) et d'autre part à comparer les résultats issus de ces appels de méthodes à ceux fournis par un oracle. L'oracle essaye de définir le comportement correct de l'objet. Dans l'environnement de test, on appelle *driver* de test l'entité qui active les méthodes de l'objet en fonction de

son jeu de test. Un métaobjet peut activer chacune des méthodes publiques d'un objet. A condition de connaître les arguments à passer à ces méthodes, il peut alors être utilisé comme driver de test de cet objet (cf. figure 37).

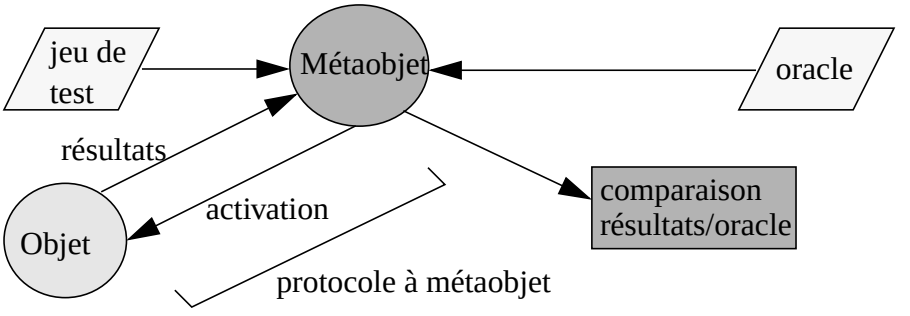

Figure 37 - Utilisation d'un métaobjet comme driver de test

L'avantage de cette approche est le fait que le métaobjet de test représente un driver de test générique. En effet, son implémentation ne dépend ni de la classe testée, ni des jeux de test. En revanche, le code ajouté à la classe pour la rendre réflexive, peut éventuellement masquer ou activer des effets de bords qui auraient été différents pour la classe originale. Par conséquent, cette approche n'est pas adaptée au test d'applications standards puisqu'elle nécessite la modification des classes de celle-ci. Puisque les classes des applications intégrées dans notre architecture sont rendues réflexives afin de bénéficier des mécanismes de tolérance aux fautes que nous proposons, elles peuvent être testées suivant cette approche.

V.1.3.2 Injection de fautes

Suivant le même principe, nous présentons ici les bases de l'utilisation de notre protocole à métaobjets pour l'injection de fautes. En effet, dans une certaine mesure, l'injection de fautes peut être comparée au test des mécanismes de tolérance aux fautes.

L'injection de faute est un moyen d'évaluation du comportement d'un système en présence de fautes. Il s'agit d'injecter des fautes dans le système afin d'observer sa réaction, sa capacité à tolérer les fautes ou ses modes de défaillance. L'injection de faute requiert donc des moyens d'action et d'observation sur le système : action pour effectivement injecter les fautes, et observation pour contrôler le comportement du système une fois les fautes injectées. Ces notions d'action et d'observation correspondent bien à celles de réification et intercession. Un tel concept a déjà été utilisé pour l'injection de fautes dans un micro-noyau [Salles *et al.* 1999] et dans des applications orientées-objet [Rosa et Martins 1998].

On peut, au sein de cette architecture, concevoir un métaobjet injecteur de fautes, qui modifie les paramètres d'appel des méthodes, modifie le séquencement de ces différents appels, ou encore, corromp l'état de l'objet. Le métaobjet observe ensuite les réactions de l'objet pour évaluer la tolérance ou la non-tolérance des fautes

injectées ou encore les modes de défaillance de l'objet. Le métaobjet peut utiliser différents séquencements d'injection pour simuler des fautes transitoires, intermittentes ou permanentes.

Le métaobjet dispose de tous les outils nécessaires à l'injection de fautes. Comme présenté en V.1.4, il est possible d'utiliser cette approche pour évaluer les métaobjets de tolérance aux fautes développés dans cette architecture.

V.1.4 Test de l'architecture

Le test de notre architecture est un élément important qui influe grandement sur la confiance que l'on porte dans les systèmes basés sur celle-ci. Ce test est actuellement à l'étude et fait l'objet d'une thèse, nous essayons toutefois ici d'en donner les principes.

Au sein de cette architecture, on peut identifier plusieurs éléments différents qui feront l'objet de tests séparés :

- l'application proprement dite, qui est indépendante de l'architecture,
- les services fournis : fabriques d'objets et de métaobjets ainsi que service de groupe, qui sont tous trois relativement indépendants du reste de l'architecture et qui peuvent être utilisés dans d'autres contextes,
- le protocole à métaobjet, dont l'implémentation dépend de l'application puisqu'il est en partie intégré aux classes qui la composent,
- et les mécanismes de tolérance aux fautes, qui dépendent d'autres éléments de l'architecture : protocole et services.

En ce qui concerne le test de l'application, celui-ci peut-être mené de façon classique suivant les approches de test orienté-objet. Il faut toutefois noter que pour prendre place au sein de notre architecture, les classes de l'application sont modifiées (pour les rendre réflexives). Il faudra donc tester les classes issues de notre compilateur et non les classes originales, afin de tester les classes effectivement utilisées. Pour cela, l'approche présentée en V.1.3, qui consiste à utiliser un métaobjet comme driver de test, peut éventuellement être utilisée.

Les différents services de l'architecture peuvent quant à eux être testés de façon totalement séparée. Ils sont, en effet, indépendants des autres éléments de l'architecture.

Des différentes entités à tester, c'est le protocole à métaobjets qui soulève le plus d'interrogations. A notre connaissance, il n'existe pas de travaux portant actuellement sur ce sujet. Bien qu'il soit défini de manière générale, l'implémentation du protocole à métaobjets dépend, dans la pratique, des classes auxquelles il est appliqué. Bien qu'une modélisation de haut-niveau soit possible et permette de l'étudier (une telle modélisation en Pi-Calcul a été étudiée dans [Marsden et Ruíz-García 1999]), chacune de ses réalisations concrètes doit, dans l'absolu, être testée.

Comme lors de sa conception, on peut, pour son test, décomposer le protocole à métaobjets en deux parties distinctes : la gestion comportementale et celle de l'état. Tester la partie comportementale consiste à s'assurer d'une part que toutes les méthodes publiques sont bien réifiées au métaobjet (réification) et d'autre part, que les méthodes originales puissent être activées par celui-ci (intercession) ; dans les deux cas il faut également s'assurer de l'empaquetage et du dépaquetage correct des arguments. Pour tester la réification, un driver de test qui active les méthodes publiques au niveau de base et un métaobjet qui observe leur réification peuvent s'avérer suffisants. En revanche, en ce qui concerne l'intercession, il est difficile d'observer l'activation des méthodes de manière non-intrusive pour la classe de base. Pour tester la gestion de l'état, nous pensons que les accesseurs peuvent être utiles : de subtiles modifications de ceux-ci à des fins d'observation permettraient d'évaluer la validité des méthodes de sauvegarde et de restauration de l´état.

Enfin, pour le test des mécanismes de tolérance aux fautes, c'est à dire des métaobjets de l'architecture, seule l'injection de fautes permet de tester leur fonctionnement. Le type de fautes et la technique d'injection dépend des hypothèses et du modèle de fautes considéré lors de la conception et de la réalisation du système. La mise en œuvre de l'injection de fautes par logiciel peut se faire en utilisant le protocole à métaobjets. En effet, comme présenté en V.1.3.2, un métaobjet possède tous les moyens d'action et d'observation nécessaires à l'injection de fautes. Un exemple d'une telle utilisation est donné en figure 38.

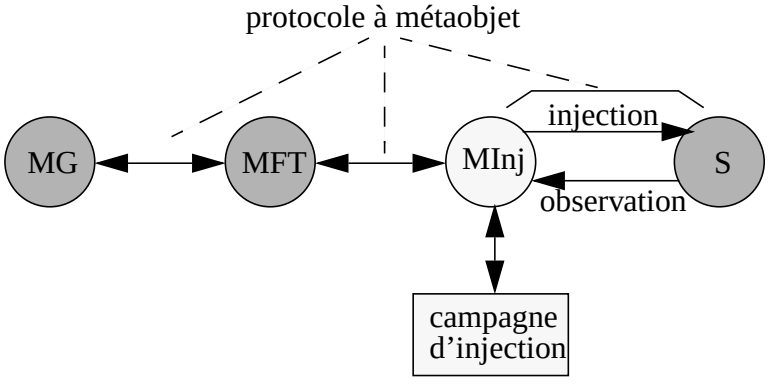

Figure 38 - Test des mécanismes de tolérance aux fautes-injection de fautes logicielles par un métaobjet

Le mécanisme présenté dans cette section, qui se base sur l'hypothèse de silence sur défaillance, ainsi que le mécanisme de détection utilisé, permettent uniquement de tolérer les fautes physiques conduisant à une défaillance par arrêt. Par conséquent la technique d'injection de fautes à utiliser ne nécessite pas, à proprement parler, de métaobjets.

V.1.5 Conclusion

Dans ce chapitre, nous avons montré les multiples intérêts du protocole à métaobjets pour mettre en œuvre des mécanismes et des techniques de sûreté de fonctionnement sur une architecture CORBA. Nous avons ainsi défini une architecture articulée autour de CORBA et du protocole à métaobjets proposé (cf. figure 39). La plate-forme sur laquelle s'appuie cette architecture est constituée d'un système d'exploitation, d'un ORB, d'un protocole de communication de groupe et enfin, d'un ou plusieurs environnements d'exécution de langage. Cette architecture, outre le protocole à métaobjets qui a été conçu et décrit dans les chapitres précédents, est composée de différents services, d'un métaniveau où se placent les différents métaobjets (les mécanismes non-fonctionnels) et de l'application. Le lien entre application et métaniveau est entretenu de façon transparente par le protocole à métaobjet.

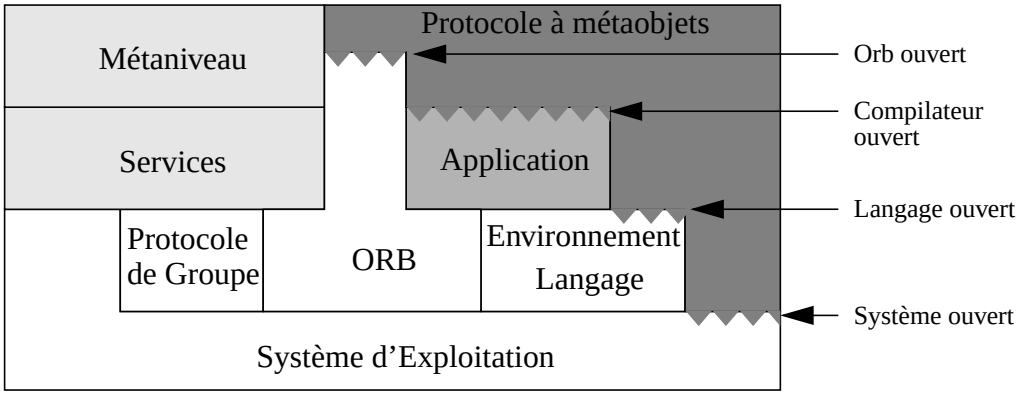

Figure 39 - L'architecture, ses composants et moyens d'extension

Du point de vue de l'application et du métaniveau, la plate-forme de l'architecture est abstraite :

- L'application s'appuie sur les mécanismes fournis par son langage d'implémentation et par CORBA.
- Les services proposés rendent indépendant le métaniveau du système d'exploitation et du protocole de groupe sous-jacents : la fabrique d'objets permet de créer des objets actifs sans se soucier du modèle d'exécution du système et le service de groupe peut aisément s'adapter à différents protocoles de communication de groupe.
- CORBA rend transparent le langage d'implémentation de l'application pour le métaniveau. Nous avons montré au chapitre précédent que le protocole à métaobjets pouvait être implémenté avec C++ et Java grâce aux compilateurs ouverts OpenC++ et OpenJava.

- Le protocole à métaobjets permet de contrôler de façon transparente le comportement et l'état des objets applicatifs au métaniveau. De plus, grâce à la notion d'interface, il est aisé de changer le métaobjet d'un objet de façon dynamique. Il suffit pour cela que le métaobjet n'ait pas d'état ou qu'il puisse le fournir au métaobjet qui le remplace, ce qui est relativement aisé pour un métaobjet de tolérance aux fautes.

Notre approche tendant à considérer une plate-forme COTS, l'architecture et le protocole à métaobjets ne dépendent que très peu de celle-ci. On peut néanmoins identifier certains points pour lesquels une adaptation à la plate-forme est nécessaire :

- La fabrique d'objets dépend du système d'exploitation puisque son rôle est de faire le lien entre objet CORBA et entité active du système. Grâce à la notion d'interface, la modification de l'implémentation de la fabrique est transparente pour ses clients.
- Le service de groupe repose sur le protocole de groupe utilisé, ici xAMp [Veríssimo et Marques 1990]. Néanmoins l'architecture du service permet de rendre ses clients indépendants de tout changement interne.
- Le protocole à métaobjet, quant à lui, dépend un peu de l'ORB utilisé. En effet, il utilise certaines fonctions de l'ORB, notamment lors de l'initialisation, qui, bien que normalisées, sont parfois quelque peu différentes selon l'ORB utilisé. En tout état de cause, les modifications nécessaires à l'utilisation d'un nouvel ORB sont minimes et très localisées.

Les différents éléments constitutifs de la plate-forme sont donc aisément interchangeables. De plus, le protocole à métaobjets peut s'adapter aux caractéristiques de ces différents éléments :

- Si le système d'exploitation est suffisamment ouvert pour fournir certaines informations relatives à l'état externe des objets (fichiers, tâches, etc.), le protocole à métaobjets peut inclure ces informations dans l'état des objets.
- Si la norme CORBA évolue, et, par exemple, que les intercepteurs (spécification CORBA 3.0) soient implémentés dans l'ORB utilisé, ils peuvent être utilisés pour le contrôle du comportement des objets applicatifs.
- Lorsque le langage utilisé pour l'implémentation des objets est réflexif, le protocole à métaobjets peut également en tirer parti, comme c'est le cas avec la sérialisation des objets en Java.
- Enfin, quand toutes les informations nécessaires aux mécanismes de tolérance aux fautes (cf. chapitre III) ne sont pas disponibles à partir des éléments de la plate-forme, la réflexivité à la compilation est utilisée pour les exhiber.
- En tout état de cause, la réflexivité à la compilation est utilisée pour assurer que certaines conventions de programmation sont respectées. Ces conventions peuvent également évoluer, en particulier pour s'adapter à certaines hypothèses que pourraient nécessiter divers éléments du système.

L'architecture et le protocole à métaobjets peuvent donc non seulement s'adapter facilement aux différentes évolutions technologiques de la plate-forme utilisée, mais sont également suffisamment ouverts pour pouvoir tirer profit de ces évolutions.

V.2 Résultats

L'implémentation partielle de l'architecture proposée permet de donner ici quelques résultats préliminaires sur l'utilisation du protocole à métaobjets. Nous proposons ici quelques relevés de performances pour plusieurs types d'activités du protocole à métaobjets : la création des objets, l'invocation de méthodes et l'obtention ou la restauration de l'état.

Ces différentes mesures ont été obtenues sur une plate-forme constituée de :
- Sun Ultra Enterprise 450 et Sun SparcStation 20,
- Solaris 2.6,
- Orbacus 3.1,
- G++ 2.8.1,
- Réseau Ethernet 100Mbits.

V.2.1 Expériences relatives à la création des objets

Dans cette architecture, deux types de mesures peuvent être faites concernant la création des objets : une mesure du temps de création du couple objet-métaobjet, à partir du serveur qui implémente l'objet lui-même, et une mesure du temps de création d'une réplique à partir de la fabrique d'objets.

V.2.1.1 Création du couple objet-métaobjet

La première de ces mesures concerne donc la création d'un objet réifié à partir d'un serveur et est à comparer avec la création du même objet mais non-réifié. En effet, la création d'un objet réifié implique, en plus de la création de l'objet lui-même, l'appel à la fabrique de métaobjets pour la création du métaobjet, ainsi que le protocole d'initialisation entre objet et métaobjet (*Meta_Startup*/*Base_Startup*).

Ces performances ont été relevées pour les deux versions de la fabrique de métaobjets décrites dans la section précédente : la première (fabrique 1) crée les métaobjets dans son propre espace d'adressage, et la seconde (fabrique 2) les crée dans des processus indépendants en utilisant la fabrique d'objets. Les résultats de ces différentes expériences sont donnés dans le tableau 32.

La création du métaobjet est relativement peu coûteuse : le coût de 7 milli-secondes environ inclut la création du métaobjet, ainsi que trois invocations CORBA (*Make-Metaobject*,*Meta_Startup* et *Base_Startup*). En revanche, il apparaît clairement que la création d'un métaobjet dans un processus indépendant, avec la fabrique 2, est beaucoup plus coûteuse, mais nécessaire pour le confinement d'erreurs

(cf. V.2.1.3). Ce coût inclut des lectures des fichiers de configuration, la création d'un nouveau processus et le lancement d'un exécutable (40 ms. environ) en plus de la création du métaobjet et des trois invocations décrites précédemment. Ce résultat peut être grandement amélioré en modifiant les fabriques d'objets et de métaobjets pour qu'elles ne lisent qu'une seule fois leurs fichiers de configuration. On peut leur ajouter une méthode pour leur faire relire ceux-ci lorsqu'ils sont modifiés, à la manière du serveur *inetd* sous Unix.

Type d'objet créé	Temps moyen de création
Objet non-réifié	4 ms.
Objet réifié (fabrique 1)	11 ms.
Objet réifié (fabrique 2)	999 ms.

Tableau 32 - Performances de la création d'objets

V.2.1.2 Création de répliques

La seconde mesure intéressante est celle de la création d'un objet (une réplique) à partir d'un client de la fabrique d'objets. En effet, ce temps inclut non seulement les opérations précédentes (création de l'objet et de son métaobjet), mais également la création d'un nouveau processus. Comme nous l'avons vu précédemment ce coût est important. Cette opération de création de réplique sera utilisée par les métaobjets de tolérance aux fautes lors du recouvrement. Le tableau 33 résume les résultats obtenus avec les deux types de fabrique de métaobjets.

Type de fabrique de métaobjet	Temps moyen de création d'une nouvelle réplique
Fabrique 1	0,97 s
Fabrique 2	2 s

Tableau 33 - Temps de création d'une réplique

Dans ces résultats, il apparaît clairement, encore une fois, que la création d'un processus indépendant par la fabrique d'objet est coûteux. Le temps de cette création est d'environ une seconde. En effet, lorsque l'on utilise la première fabrique de métaobjet, un seul processus indépendant est créé, celui de l'objet. En revanche avec la deuxième version, deux processus sont créés, le premier pour l'objet et le second pour le métaobjet.

Les temps donnés ici (et en V.2.1) pour la fabrique 2 dépendent donc en grande partie du temps de création d'une entité d'exécution sur le système sous-jacent. Le protocole à métaobjet est, dans ce cas, relativement indépendant des résultats obtenus.

V.2.1.3 Des fabriques de métaobjets

Dans les deux paragraphes précédents, il apparaît clairement que le choix du type de fabrique de métaobjet influe grandement sur les performances de création des objets, que ces objets soient de simples serveurs ou qu'ils soient répliqués. L'activité de création d'objet est utilisée à l'initialisation du système ainsi qu'en cas de recouvrement.

Le deuxième type de fabrique de métaobjet, qui crée chacun des métaobjets dans un processus indépendant, est nécessaire lorsque les hypothèses de fautes et les modes de défaillance du système considérés impliquent le confinement des erreurs dans chacune des entités. En effet, le système d'exploitation, ici Unix, n'assure le confinement d'erreur qu'entre les différents processus. En revanche, si la plate-forme (système ou ORB) proposait une granularité plus fine, en termes d'objets par exemple, la création de processus indépendants serait alors inutile et la création des objets en serait alors plus performante.

Si l'hypothèse de silence sur défaillance par arrêt de la machine est considérée, le confinement d'erreur intra-site n'est pas nécessaire et on peut se contenter d'utiliser le premier type de fabrique de métaobjets, qui est bien plus performant. En effet, cette hypothèse permet d'assurer un confinement inter-site des erreurs, suffisant pour tolérer ce type de fautes.

En revanche, si le confinement d'erreur intra-site est nécessaire, la deuxième fabrique doit alors être utilisée. On pourra néanmoins rendre le processus de recouvrement moins handicapant en utilisant des mécanismes de tolérance aux fautes qui autorisent le clonage en parallèle avec l'exécution du système en mode dégradé. De tels algorithmes ont été développés pour des systèmes de transport hautement critiques en termes de vies humaines [Essame 1998].

V.2.2 Expériences relatives aux invocations

Alors que la création d'objets et de répliques est une activité essentiellement utilisée lors de l'initialisation et du recouvrement du système, l'invocation de méthode, quant à elle, est l'activité principale lors du fonctionnement normal, c'est-à-dire en l'absence de faute. Les performances de ces invocations influent donc grandement sur les performances globales du système. Afin d'illustrer les différentes caractéristiques qui peuvent influer sur le temps d'invocation d'une méthode, nous avons mené nos expérimentations sur trois méthodes différentes :

- La première est une méthode simple et rapide, elle prend un entier en paramètre d'entrée, l'incrémente et retourne le résultat de cette opération.
- La seconde méthode implique des entrées/sorties coûteuses, elle reçoit une chaîne de caractère en entrée et écrit 200 fois cette chaîne dans un fichier.
- Enfin, la troisième possède de nombreux arguments de type complexe : une chaîne de caractère, un flottant, un entier long et enfin une référence d'objet CORBA. Tous ces paramètres sont en entrée/sortie.

Nous comparons, pour chacune de ces trois méthodes, les temps d'invocation d'un client à un serveur CORBA non-réifié au temps d'une invocation réifiée qui passe par un métastub et un métaobjet pour aller du client au serveur. Les résultats obtenus avec les deux types de fabrique de métaobjets (la fabrique 1 crée des métaobjets dans son propre espace d'adressage et la fabrique 2 dans un processus indépendant) sont donnés au tableau 34.

	Objet non-réifié	Objet réifié (fabrique 1)	Objet réifié (fabrique 2)
Méthode légère	0,5 ms.	2,5 ms. (+2)	5,1 ms. (+4,6)
Méthode lourde	4,3 ms.	6,3 ms. (+2)	9 ms. (+4,7)
Méthode avec de nombreux arguments	3,1 ms.	5,7 ms. (+2,6)	9,4 ms. (+6,3)

Tableau 34 - Performances des invocations à partir du client

Il ressort de ces différentes expériences que le surcoût dû à la réification/interception des méthodes dépend en grande partie des arguments de la méthode invoquée. En effet, le protocole implique pour chaque invocation deux empaquetages et deux dépaquetages supplémentaires des arguments. En effet, par rapport à une invocation simple, le nombre d'intervenants est doublé (client, métastub, métaobjet et objet), le nombre d'invocations est triplé (du client au métastub, du métastub au métaobjet et enfin du métaobjet à l'objet), le traitement des arguments à chaque invocation a donc une part importante dans le coût de la réification. Plus les arguments sont nombreux et complexes, plus le coût de leur gestion est important. On notera que ceci est également le cas avec les objets non-réifiés.

Les métaobjets, lorsqu'ils sont créés dans un processus indépendant sont moins performants que lorsqu'ils sont créés dans la fabrique de métaobjets. Cette contre-performance est ici amplifiée par le fait que les métastubs sont également créés dans des processus indépendants. Ce coût est néanmoins le prix à payer pour assurer un meilleur confinement des erreurs. En effet, pour ce faire, la défaillance d'un métaobjet ne doit pas entraîner celle des autres entités. Il est clair que plus les hypothèses de fautes du système sont faibles, plus le prix à payer pour la tolérance aux fautes est important.

Afin d'améliorer ces performances, on pourra utiliser le premier type de fabrique de métaobjets, qui n'assure pas le confinement des erreurs entre ceux-ci. Une deuxième solution plus radicale consisterait à créer directement les métaobjets dans l'espace de l'objet et les métastubs dans l'espace du client. L'encapsulation, c'est-à-dire le confinement d'erreurs serait encore moins fort, le changement dynamique de métaobjet serait rendu plus complexe, mais les performances du système en l'absence de fautes n'en seraient que meilleures. Enfin, comme nous l'avons dit dans les paragraphes précédents, la solution optimale à ce problème de confinement

consiste en l'utilisation de mécanismes de protection fournis par l'ORB, le système ou le matériel sous-jacent qui ne pourraient, alors, être considérés comme des boîtes noires.

V.2.3 Expériences relatives à l'état des objets

Les différentes actions relatives à l'état des objets sont l'obtention et la restauration de celui-ci. La complexité des mécanismes mis en œuvre varie grandement en fonction de celle de la classe de l'objet considéré : le nombre et le type des attributs. C'est pourquoi, nous avons mené ces expériences sur quatre classes différentes, classées par ordre croissant de complexité :

- une classe *A*, qui contient uniquement un entier, un flottant et une chaîne de caractères.
- une classe *B*, qui contient un tableau d'entiers, un tableau de flottants et un tableau de chaînes de caractères. Chaque tableau possède 10 éléments.
- une classe *C*, qui contient les mêmes tableaux que la classe *B*, mais dont chaque tableau possède 200 éléments.
- et une classe *D*, qui représente une liste chaînée de 100 objets de type *A*.

Pour chacune de ces classes, nous avons mesuré le temps d'obtention et de restauration de l'état complet, via les méthodes *Save/Restore_FullState*, ainsi que de l'état partiel avec les méthodes *Save/Restore_PartialState*. En ce qui concerne l'état partiel, avant chaque sauvegarde de l'état de l'objet, celui-ci a modifié ses attributs en proportion variable (de 0% à 100%). Afin de pouvoir comparer ces résultats, les mêmes expériences ont été menées avec le processus de sérialisation du langage Java sur des classes strictement équivalentes. Les tableaux 35 et 36 résument ces différents résultats.

Pour les classes relativement simples (A, B et C), il semble qu'il ne soit pas intéressant d'utiliser les méthodes relatives à l'état partiel. Etant donné que ces classes sont constituées d'attributs de type simple (ou de tableaux de type simple), le coût de la gestion de l'état partiel est trop important en comparaison de celui d'une copie des éléments de type simple.

Par contre, pour la classe D qui est la plus complexe et qui, par composition, est constituée d'objets et dont les méthodes de sauvegarde et de restauration de l'état sont récursives, considérer l'état partiel est beaucoup plus intéressant. En effet, ici, le coût de la création (et de la destruction) des objets internes, ainsi que le coût de la récursivité, sont suffisamment importants, par rapport au coût de la gestion de l'état partiel, pour que celle-ci soit efficace. On notera également que pour cette classe, le processus de sérialisation de Java est relativement efficace. Java tire, dans ce cas, un grand bénéfice de la réflexivité structurelle dont il dispose.

	A	B	C	D
Complet	0,02	0,08	1,4	1359
Partiel 0%	0,02	0,25	1,6	20
Partiel 10%	0,02	0,5	9,2	36
Partiel 20%	0,02	0,8	16,7	75
Partiel 30%	0,02	1,3	20,6	140
Partiel 40%	0,05	1,5	32	230
Partiel 50%	0,05	2,1	36	344
Partiel 60%	0,05	2,3	40	499
Partiel 70%	0,2	2,5	58	652
Partiel 80%	0,2	2,7	62	838
Partiel 90%	0,2	3,8	66	1064
Partiel 100%	0,2	4	70	1370
Java	2	3	4	157

Tableau 35 - Performances de l'obtention de l'état (en milli-secondes)

	A	B	C	D
Complet	0,006	0,027	0,44	390
Partiel 0%	0,17	1,9	34	509
Partiel 10%	0,17	1,9	34	518
Partiel 20%	0,17	1,9	34	532
Partiel 30%	0,17	1,9	34	538
Partiel 40%	0,17	1,9	34	543
Partiel 50%	0,17	1,9	34	560
Partiel 60%	0,17	1,9	34	576
Partiel 70%	0,17	1,9	34	598
Partiel 80%	0,17	1,9	34	625
Partiel 90%	0,17	1,9	34	640
Partiel 100%	0,17	1,9	34	673
Java	1	4	20	204

Tableau 36 - Performances de la restauration de l'état (en milli-secondes)

Pour conclure, il semble que les méthodes d'obtention et de restauration de l'état complet, générées grâce à la réflexivité à la compilation, soient relativement efficaces lorsqu'on les compare au processus de sérialisation fourni par Java. Les techniques relatives à l'état partiel, en revanche, ne sont intéressantes que lorsque la classe considérée est complexe. L'utilisateur devra donc sélectionner la technique que les mécanismes de tolérance aux fautes utiliseront pour la gestion de l'état de ses objets, en fonction de chaque classe de l'application et de son profil d'utilisation.

V.3 Conclusion

Ce chapitre montre comment un protocole à métaobjet permet de définir une architecture flexible pour la mise en œuvre de mécanismes de tolérance aux fautes. Nous avons illustré les propriétés attendues et l'impact sur les performances aux travers d'exemples simples.

Cette architecture se base sur une plate-forme abstraite constituée d'un système d'exploitation, d'un ORB, de protocoles de communication de groupe et d'un ou plusieurs supports d'exécution de langage. Les autres éléments qui constituent l'architecture sont : des services CORBA qui permettent d'abstraire la plate-forme vis-à-vis des métaobjets, l'application pour qui l'architecture est rendue transparente grâce aux abstractions fournies par CORBA, de métaobjets de tolérance aux fautes et, enfin, du protocole à métaobjets qui permet aux métaobjets de contrôler l'application de façon transparente, et ce, quel que soit son langage d'implémentation, C++ ou Java.

Bien que l'architecture soit conçue de manière indépendante de la plate-forme, certains éléments de l'architecture dépendent légèrement de celle-ci. C'est le cas notamment de certains services comme la fabrique d'objets ou du service de groupe. Néanmoins, grâce à la notion d'interface, l'implémentation de ces services peut être modifiée pour s'adapter à une nouvelle plate-forme, sans toutefois que ces modifications se répercutent sur le reste de l'architecture. Clairement, les modifications nécessaires à un portage sur une plate-forme différente sont extrêmement limitées.

Cette propriété forte d'indépendance de l'architecture permet, en outre, à celle-ci d'être fortement évolutive. En effet, le protocole à métaobjet, qui est au cœur de l'architecture, peut être facilement adapté pour tirer parti de toute nouvelle fonctionnalité de la plate-forme. Par exemple, si le système d'exploitation est capable de fournir de l'information à propos de l'ordonnancement des tâches, ou de l'état du système de fichier, le protocole à métaobjets peut prendre cette information en compte dans l'état des objets. Lorsque la norme CORBA 3.0 sera implémentée dans les ORBs du commerce, les intercepteurs pourront être utilisés pour contrôler le comportement des objets de l'application. Lorsque le langage d'implémentation de l'application est, dans une certaine mesure, réflexif, cette réflexivité est mise à profit dans le protocole à métaobjet, comme c'est le cas de la sérialisation des objets Java.

De nouveaux protocoles de communication de groupe, avec des propriétés plus fortes, ou une meilleure couverture des hypothèses, peuvent être utilisés. Enfin, il est aisé de développer de nouveaux mécanismes de tolérance aux fautes au sein de métaobjets. Ce développement peut ré-utiliser certaines parties de la conception et du code des mécanismes déjà développés. De plus, l'architecture permet de changer de métaobjet, et donc de mécanisme de tolérance aux fautes, de façon dynamique, pendant le cycle de vie de l'application.

Dans la seconde section de ce chapitre, nous avons proposé quelques résultats préliminaires issus de l'implémentation partielle disponible à ce jour. Ces résultats ont porté sur différentes caractéristiques du protocole à métaobjets et de l'architecture telles que la création de métaobjets et de répliques d'objets de l'application, les invocations entre un client et un serveur, et enfin, sur les différentes méthodes d'obtention de l´état des objets. Ces résultats nous ont permis d'illustrer une caractéristique importante de l'architecture relative aux hypothèses de fautes du système. En effet, au vu des résultats, il apparaît clairement que la création des métaobjets dans des processus indépendants a un impact important sur les performances de l'application, en l'absence et en présence de fautes. Cette création de métaobjets dans des processus indépendants est nécessaire pour le confinement d'erreur lorsque l'hypothèse de silence sur défaillance n'est pas prise, ou lorsque le système d'exploitation ne permet pas, par lui-même, le confinement d'erreur à une granularité plus fine que les processus, dans des objets, par exemple. Grâce à ces résultats, nous avons également identifié que les algorithmes d'obtention et de restauration de l'état partiel sont très intéressants lorsque les objets considérés sont complexes. En effet, dans ce cas, et dans ce cas seulement, les performances liées aux méthodes de gestion de l'état partiel sont bien meilleures que celles de la gestion de l'état complet.

Le développement du mécanisme de réplication passive, dans la première section, nous a permis d'illustrer les propriétés et bénéfices tirés de l'utilisation de notre protocole à métaobjet, et de la réflexivité à la compilation, pour l'intégration de mécanismes de tolérance aux fautes à une application, de façon transparente. Le protocole à métaobjets et l'architecture sont basés sur une plate-forme COTS et peuvent néanmoins être facilement adaptés lorsque cette plate-forme fournit un certain niveau de réflexivité ou plus simplement d'introspection.

Au delà de la tolérance aux fautes, nous avons également montré que le protocole et l'architecture pouvaient être utilisés pour le développement d'autres mécanismes de sûreté de fonctionnement. En effet, le protocole à métaobjets est suffisamment général pour permettre le test des objets de l'application, ou encore l'injection de fautes pour la validation des mécanismes développés dans cette architecture.

CONCLUSION GÉNÉRALE

L'originalité du travail présenté dans ce document consiste essentiellement en l'utilisation du concept de langage ouvert, et en particulier de réflexivité à la compilation, afin d'implémenter un protocole à métaobjets dédié à la tolérance aux fautes d'objets CORBA. En effet, certaines études ont montré que les protocoles à métaobjets généraux existant ne sont pas satisfaisants dans ce contexte : manque d'information réifiée, staticité du lien entre objet et métaobjet, etc. Nous avons donc conçu et réalisé notre propre protocole à métaobjets en utilisant la réflexivité à la compilation. Pour ce faire, notre démarche a été la suivante :

Nous avons, dans un premier temps, étudié différents mécanismes de tolérance aux fautes afin d'identifier les informations nécessaires à leur fonctionnement. On peut classer ces informations en quatre catégories distinctes :

- L'observation et le contrôle du comportement des objets, qui consistent essentiellement en la réification et la possibilité d'activation des méthodes de l'objet et qui incluent les paramètres d'appel et de retour des invocations.
- Le contrôle de l'état des objets, c'est-à-dire la possibilité de sauvegarder et de restaurer l'état des objets. On notera que l'on distingue deux parties qui constituent l'état d'un objet : l'état interne qui regroupe les valeurs des attributs de l'objet et qui est encapsulé par celui-ci, et l'état externe, qui est plus ou moins caché dans les couches inférieures du système et que l'objet peut difficilement contrôler.
- Le contrôle des relations client/serveur qui peuvent exister entre les objets CORBA, ce contrôle se faisant par le biais des références CORBA sur le serveur.
- Enfin, le contrôle du lien objet/métaobjet, qui doit être dynamique, c'est-à-dire qu'un objet doit pouvoir changer dynamiquement de métaobjet et vice-versa.

Ensuite, nous avons parcouru les différents moyens d'obtention de ces informations, en fonction de la plate-forme et du langage utilisés, et plus particulièrement de leur degré d'ouverture, ou de réflexivité. Cette étude a notamment porté sur les langages C++, qui n'est pas du tout réflexif, et Java, qui fournit de la réflexivité structurelle mais non comportementale, ainsi que sur la plate-forme CORBA 3.0, qui, quand elle sera implémentée, devrait supporter l'interception de l'activité comportementale des objets. Il apparaît clairement, dans cette étude, que l'utilisation de la réflexivité à la compilation permet de s'affranchir, dans une certaine mesure, des limites imposées par une plate-forme et un langage COTS, c'est-à-dire "boite-noire". En effet, l'utili-

sation d'un compilateur réflexif permet d'identifier, à la compilation, les informations pertinentes et de modifier l'application afin d'exhiber ces informations au moment de l'exécution. En outre, l'utilisation de la réflexivité à la compilation permet d'assurer le filtrage du code source de l'application et d'appliquer des conventions de programmations. Ce processus de filtrage n'est évidemment possible qu'au moment de la compilation.

Le protocole à métaobjets, ainsi défini et réalisé, permet alors d'implémenter des mécanismes non-fonctionnels, en l'occurrence de tolérance aux fautes, de façon transparente et bien séparée de l'application. En effet, ce protocole à métaobjets permet aux mécanismes, implémentés dans les métaobjets, de contrôler le comportement et l'état interne des objets. Il prend également en compte les relations clients-serveurs qui existent entre les différents objets CORBA de l'application. Enfin, il utilise judicieusement la notion d'interface afin de permettre aux métaobjets d'être échangés dynamiquement pendant l'exécution de l'application. Cette dynamicité permet d'adapter le système aux différentes conditions d'exécution et hypothèses de fautes, "en direct" lors de l'exécution du système.

L'architecture proposée pour mettre en œuvre ce protocole à métaobjets est également flexible et évolutive. Les couches inférieures du système sont abstraites par l'utilisation de services CORBA. De plus. l'architecture est adaptable à différentes hypothèses de fautes du système : elle peut fournir différents niveaux de confinement d'erreur, à un coût variable en fonction des hypothèses de fautes.

Nous avons, toutefois, identifié certaines limitations dues à l'utilisation d'une plate-forme COTS. En effet, certaines informations, nécessaires aux mécanismes de tolérance aux fautes, ne sont pas disponibles au niveau du langage. Ces informations peuvent être d'ordre sémantique, par exemple, lorsque l'application, ou une partie d'elle, est indéterministe. D'autres informations, indisponibles au niveau langage, concernent essentiellement l'état externe des objets et résident dans les couches sous-jacentes, systèmes d'exploitation ou ORB. C'est le cas, par exemple, des informations concernant les fichiers ouverts ou encore les différentes tâches de l'application et les changements de contexte opérés entre celles-ci par l'ordonnanceur du système.

Résoudre ces problèmes consisterait, d'une part, en l'utilisation de réfléxivité déclarative par l'utilisateur pour identifier certaines informations sémantiques, et, d'autre part, utiliser une plate-forme ouverte afin d'obtenir les informations que contiennent les couches inférieures. En effet, la tolérance aux fautes est un concept transversal au système et à l'application : elle nécessite des informations issues de toutes les couches. La solution idéale est donc d'utiliser une plate-forme réflexive de bout-en-bout, où chaque composant peut réifier l'information qu'il contient aux entités en faisant la demande.

Le protocole à métaobjets, défini ici, permet d'unifier les relations entre les différents composants du système et, comme nous l'avons montré, peut en outre s'adapter facilement pour tirer parti d'éventuels nouveaux composants réflexifs. Il offre, en outre, des propriétés très intéressantes : transparence pour l'utilisateur, adaptation au système, composabilité et réutilisation des mécanismes.

Cette approche fournit donc une solution générale au problème de la tolérance aux fautes par protocole à métaobjets dans CORBA, dont l'implémentation peut varier en fonction du degré d'ouverture du système, en partant d'une «boîte-noire» ou COTS vers une «boîte-grise» ou réflexive.

BIBLIOGRAPHIE

[Agha 1990] G. Agha, "Concurrent Object Oriented Programming", *Communications of the ACM*, vol. 33(9), pp. 125-141, 1990.

[Agha *et al.* 1993] G. Agha, S. Frolund, R. Panwar et D. Sturman, "A Linguistic Framework for Dynamic Composition of Dependability Protocols", in *DCCA-3*, pp. 197-207, 1993.

[Balter *et al.* 1990] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville et G. Vandome, "Architecture and Implementation of Guide, an Object-Oriented Distributed System", *Computing Systems*, vol. 4(1/Winter 1991), pp. 31-67, 1990.

[Birman *et al.* 1990] K. P. Birman, R. Cooper, T. A. Jospeh, K. P. Kane, F. Schmuck et M. Wood, "ISIS-a distributed programming environment", Cornell University, Cornell, 1990.

[Blair *et al.* 1998] G. S. Blair, G. Coulson, P. Robin et M. Papathomas, "An Architecture for Next Generation Middleware", in *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, UK, 1998.

[Booch 1994] G. Booch, *Object-Oriented Analysis and Design With Applications*, Addison-Wesley, ISBN 0805353402, 1994.

[Brant *et al.* 1998] J. Brant, B. Foote, R. E. Johnson et D. Roberts, "Wrappers to the Rescue", in *ECOOP'98*, Brussels, Belgium, pp. 396-417, 1998.

[Campbell *et al.* 1993] R. H. Campbell, N. Islam, D. Raila et P. Madany, "Designing and Implementing Choices: An Object-Oriented System in C++", *Communications of the ACM*, vol. 36(9), pp. 117-126, 1993.

[Chandra et Toueg 1996] T. Chandra et S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, vol. 43(2), pp. 225-267, 1996.

[Chiba 1995] S. Chiba, "A Metaobject Protocol for C++", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, Austin, Texas, USA, pp. 285-299, 1995.

[Chiba et Masuda 1993] S. Chiba et T. Masuda , “Designing an Extensible Distributed Language with a Meta-Level Architecture”, in *European Conference on Object Oriented Programming (ECOOP'93)*, pp. 482-501, 1993.

[Consel et Marlet 1998] C. Consel et R. Marlet, “Architecturing Software Using a Methodology for Language Development”, in *10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP/ALP '98)*, Pise, Italie, pp. 170-194, 1998.

[Cristian *et al.* 1995] F. Cristian, H. Aghili et D. Dolev, “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”, in *Information and Computation*, vol. 118. Ed.^Eds., pp. 158-179, 1995.

[Detlefs *et al*. 1988] D. Detlefs, M. P. Herlihy et J. M. Wing, “Inheritance of Synchronization and Recovery Properties in Avalon/C++”, *ACM Computer*, vol. 21(12), pp. 57-69, 1988.

[Dominus 1998] M.-J. Dominus, “Perl: Not Just for Web Programming”, *IEEE Software*, vol. 15(1), pp. 69-74, 1998.

[Elnozahy 1993] E. N. Elnozahy, “Manetho: Fault-Tolerance in Distributed Systems using Rollback-Recovery and Process Replication”, PhD Thesis, Rice University, 1993.

[Essame 1998] D. Essame, “Tolérance aux fautes dans les systèmes critiques : application au pilotage des lignes de métro automatisées”, Thèse de Doctorat, Institut National Polytechnique, LAAS-CNRS, Toulouse, 1998, Rapport LAAS No 98414, pp.:160.

[Fabre et Pérennou 1998] J.-C. Fabre et T. Pérennou, “A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach”, *IEEE Transactions on Computers, Special issue on Dependability of Computing Systems*, vol. 47(1), pp. 78-95, 1998.

[Felber *et al*. 1996] P. Felber, B. Garbinato et R. Guerraoui, “The Design of a CORBA Group Communication Service”, in *IEEE Symposium on Reliable Distributed Systems*, pp. 150-159, 1996.

[Fetzer et Cristian 1997] C. Fetzer et F. Cristian, “Fail-Awareness: An Approach to Construct Fail-Safe Applications”, in *27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, Seattle, WA, USA, pp. 282-291, 1997.

[Fischer *et al.* 1985] M. J. Fischer, N. Lynch et M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Processus”, *Journal of the ACM*, vol. 32(2), pp. 374-382, 1985.

[Gamma *et al.* 1995] E. Gamma, R. Helm, R. Johnson et J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Garbinato *et al.* 1995] B. Garbinato, R. Guerraoui et K. Mazouni, "Implementation of the GARF Replicated Objects Platform", *Distibuted Systems Engineering Journal*, vol. 2, pp. 14-27, 1995.

[Golm et Kleinoder 1999] M. Golm et J. Kleinoder, "Jumping to the Meta Level: Behavioral Reflection can be Fast and Flexible", in *Reflection'99*, Saint-Malo, France, pp. 22-39, 1999.

[Gosling *et al.* 1996] J. Gosling, B. Joy et G. L. J. Steele, *The Java Language Specification*, ISBN 0201634511, 1996.

[Guerraoui *et al.* 1998] R. Guerraoui, P. Felber, B. Garbinato et K. Mazouni, "System Support for Object Groups", in *OOPSLA'98*, Vancouver, British Columbia, Canada, pp. 244-258, 1998.

[Harel 1996] D. Harel, "Statecharts: Past, Present and Future", in *Theory and Practice of Informatics, Seminar on Current Trends in Theory and Practice of Informatics*, vol. 23, *Lecture Notes in Computing Science*, Springer-Verlag, Ed., Springer-Verlag, 1996.

[Hill 1993] D. R. C. Hill, *Analyse Orientée Objets et Modélisation par Simulation*, Addison-Wesley, 1993.

[Hoare 1972] C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, vol. 1, pp. 271-281, 1972.

[Honda et Tokoro 1992] Y. Honda et M. Tokoro, "Soft Real-Time Programming through Reflection", in *International Workshop on New Models for Software Architecture: Reflection and Meta-Level Architecture*, Tokyo, Japan, pp. 12-23, 1992.

[Huang *et al.* 1995] Y. Huang, C. Kintala et Y.-M. Wang, "Software Tools and Libraries for Fault Tolerance", *IEEE Technical Comitee on Operating Systems and Application Environments*, vol. 7(4), pp. 5-9, 1995.

[Hurfin *et al.* 1998] M. Hurfin, A. Mostefaoui et M. Raynal, "Consensus in Asynchronous Systems Where Processes Can Crash and Recover", in *IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, pp. 280-286, 1998.

[IONA 1997] IONA, "Orbix 2.2 Programming Guide", IONA Technologies Ltd, 1997.

[ISO 1998] ISO, “Programming languages - C++”, ANSI, ISO/IEC 14882-1998, 1998.

[Jalote 1994] P. Jalote, *Fault tolerance in distributed systems*, Englewood Cliffs, N.J., Prentice Hall, ISBN 0133013677, 1994.

[Kasbekar *et al.* 1999] M. Kasbekar, C. R. Das, S. Yajnik, R. Klemm et Y. Huang, “Issues in the Design of a Reflective Library for Checkpointing C++ Objects”, in *SRDS'99*, 1999.

[Kiczales *et al.* 1991] G. Kiczales, J. des Rivières et D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, ISBN 0-262-61074-4, 1991.

[Killijian *et al.* 1998a] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-García et S. Chiba, “Development of a Metaobject Protocol for Fault-Tolerance Using Compile-Time Reflection”, in *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, pp. 61-65, 1998a.

[Killijian *et al.* 1998b] M.-O. Killijian, J.-C. Fabre, J.-C. Ruiz-García et S. Chiba, “A Metaobject Protocol for Fault-Tolerant CORBA Applications”, in *IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, USA, pp. 127-134, 1998b.

[Killijian *et al.* 1999] M.-O. Killijian, J.-C. Ruiz-García et J.-C. Fabre, “Using Compile Time Reflection for Object State Capture”, in *Reflection'99*, Saint-Malo, France, pp. 150-152, 1999.

[Kim et Popek 1997] T. H. Kim et G. J. Popek, “Frigate: An Object-Oriented File System for Ordinary Users”, in *3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97)*, Portland, 1997.

[Kleene 1989] S. E. Kleene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

[Koo et Toueg 1987] R. Koo et S. Toueg, “Checkpointing and Rollback-Recovery for Distributed Systems”, *IEEE Transactions on Software Engineering*, vol. SE-13(1), pp. 23-31, 1987.

[Laprie *et al.* 1996] J. C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac et P. Thévenod, *Guide de la sûreté de fonctionnement*, 2ème édition, Cépaduès Éditions, ISBN 2-85428-382-1, 1996.

[Larousse 1972] Larousse, *Petit Larousse en Couleurs*, Larousse, 1972.

[Lieberman 1986] H. Lieberman, “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”, in *OOPSLA'86*, 1986.

[Liskov et Zilles 1974] B. H. Liskov et S. N. Zilles, “Programming with Abstract Data Types”, *SIGPLAN notices*, vol. 9(4), pp. 50-59, 1974.

[Long *et al.* 1991] J. Long, W. K. Fuchs et J. A. Abaham, “Implementing Forward Recovery using Checkpointing in Distributed Systems”, in *2nd IFIP Working Conference on Dependable Computations for Critical Applications*, pp. 20-27, 1991.

[Maes et Nardi 1988] P. Maes et D. Nardi, “Meta-Level Architectures and Reflection”, North-Holland. Ed.. , Elsevier Science Publishers B.V., 1988.

[Maffeis et Schmidt 1997] S. Maffeis et D. C. Schmidt, “Constructing Reliable Distributed Communication Systems with Corba”, *IEEE Communications Magazine*, vol. 14(2), pp. 6, 1997.

[Marsden et Ruíz-García 1999] E. Marsden et J.-C. Ruíz-García, “Description Formelle d'un Protocole à Métaobjets”, LAAS-CNRS, Toulouse, 99314, Juillet 1999.

[Masuhara *et al.* 1996] H. Masuhara, S. Matsuoka et A. Yonezawa, “Implementing Parallel Language Constructs using a Reflective Object Oriented Language”, in *Reflection'96*, San-Francisco, CA, USA, pp. 79-104, 1996.

[Meyer 1997] B. Meyer, *Object-Oriented Software Construction*, 2nd, Upper Saddle River, N.J., Prentice Hall, 1997.

[Moser et Melliar-Smith 1997] L. E. Moser et P. M. Melliar-Smith, “The Interception Approach to Reliable Distributed CORBA Objects”, in *3rd USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Or., USA, pp. 245-248, 1997.

[Moser *et al.* 1996] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia et C. A. Lingley-Papadopoulos, “Totem: A Fault-Tolerant Multicast Group Communication System”, *Communications of the ACM*(April), 1996.

[Narasimhan *et al.* 1999] P. Narasimhan, L. E. Moser et P. M. Melliar-Smith, “Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications”, in *18th Symposium on Reliable Distributed Systems (SRDS'99)*, Lausanne, Suisse, 1999.

[Netzer et Weaver 1994] R. H. B. Netzer et M. H. Weaver, “Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs”, in *SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

[OMG 1997] OMG, “Meta Object Factory (MOF) Specification”, OMG, ad/97-08-14, 1997.

[OMG 1998a] OMG, “CORBA/IIOP 2.2 Specification”, , 98-07-01, 1998a.

[OMG 1998b] OMG, “Externalization Service Specification”, OMG, orbos/98-12-16, 1998b.

[OMG 1998c] OMG, “Fault Tolerant CORBA Using Entity Redundancy”, OMG, orbos/98-04-01, 1998c.

[OMG 1998d] OMG, “Persistent Object Service Specification”, OMG, orbos/97-12-12, 1998d.

[OMG 1998e] OMG, “Portable Interceptors RFP”, OMG, orbos/98-09-11, 1998e.

[OMG 1999] OMG, “Portable Interceptors”, Eternal Systems, Inc, Expersoft Corporation , Sun Microsystems Inc., Initial RFP Submission, orbos/99-04-07, April, 26 1999.

[Orfali *et al.* 1996] R. Orfali, D. Harkey et J. Edwards, *The Essential Client/Server Survival Guide*, 2nd, New York, Wiley, ISBN 0471153257, 1996.

[Pérennou 1997] T. Pérennou, “Une Architecture à Métaobjets pour Systèmes Répartis Tolérant les Fautes”, Thèse de Doctorat, Institut National Polytechnique de Toulouse, LAAS-CNRS, Toulouse, 1997, Rapport LAAS No 97011, pp.:168.

[Plank 1997] J. S. Plank, “An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance”, University of Tenessee, Department of Computer Science, Knoxville, Technical Report, UT-CS-97-372, July 1997.

[Plank *et al.* 1995] J. S. Plank, M. Beck, G. Kingsley et K. Li, “Libckpt: Transparent Checkpointing under Unix”, in *Usenix Winter 1995 Technical Conference*, New Orleans, LA, USA, pp. 213-223, 1995.

[Powell 1991] D. Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Berlin, Allemagne, Springer-Verlag, 1991.

[Powell 1992] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, in *Twenty-Second International Symposium on Faut Tolerant Computing (FTCS'22)*, Boston, Massachusetts, USA, pp. 386-395, 1992.

[Powell 1998] D. Powell, “Distributed Fault Tolerance : A Short Tutorial”, in *1998 IFIP International Workshop on Dependable Computing and its Applications (DCIA'98)*, Johannesburg (Afrique du Sud), pp. 1-12, 1998.

[Rabejac 1995] C. Rabejac, “Auto-surveillance logicielle pour applications critiques : méthode et mécanismes”, Doctorat, Institut National Polytechnique, TSF / LASS-CNRS, Toulouse, 1995, 95449, pp.:145.

[Rogerson 1997] D. Rogerson, *Inside COM: Microsoft's Component Object Model*, Microsoft Press, 1997.

[Rosa et Martins 1998] A. A. Rosa et E. Martins, "Using Reflective Programming to Inject Faults into Object-Oriented Systems", in *IFIP International Workshop on Dependable Computing and Its Applications (DCIA'98)*, Johannesburg, Afrique du Sud, pp. 227-236, 1998.

[Ruiz-García *et al.* 1998] J.-C. Ruiz-García, M.-O. Killijian, J.-C. Fabre et S. Chiba, "Optimized Object State Chekpointing using Compile-Time Reflection", in *Workshop on Embedded Fault-Tolerant Systems (EFTS'98)*, Boston, MA, USA, pp. 46-48, 1998.

[Rumbaugh *et al.* 1998] J. Rumbaugh, I. Jacobson et G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, ISBN 020130998X, 1998.

[Salles *et al.* 1999] F. Salles, M. Rodríguez, J.-C. Fabre et J. Arlat, "MetaKernels and Fault Containment Wrappers", in *29th International Symposium on Fault-Tolerant Computing (FTCS'29)*, Madison, Wisconsin, USA, pp. 22-29, 1999.

[Schmidt et Vinoski 1997] D. C. Schmidt et S. Vinoski, "Object Interconnections, Object Adapters: Concept and Terminology", *SIGS C++ Report*, vol. 9(11), 1997.

[Schneider 1993] F. B. Schneider, "What Good are Models and What Models are Good", in *Distributed systems*, *ACM Press frontier series*, S. Mullender, Ed., 2nd ed. New York, ACM Press; Addison-Wesley, pp. xvi, 595, 1993.

[Sekiguchi *et al.* 1999] T. Sekiguchi, H. Masuhara et A. Yonezawa, "A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation", in *Third International Conference on Coordination Models and Languages (Coordination'99)*, Amsterdam, Pays-Bas, 1999.

[Shlaer et Mellor 1991] S. Shlaer et S. J. Mellor, *Object Lifecycles : Modeling the World in States*, Yourdon, ISBN 0136299407, 1991.

[Shrivastava *et al.* 1991] S. K. Shrivastava, G. N. Dixon et G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, vol. 8(1), pp. 66-73, 1991.

[Slye et Elnozahy 1996] J. H. Slye et E. N. Elnozahy, "Supporting Nondeterministic Execution in Fault-Tolerant Systems", in *26th Annual International Symposium on Fault-Tolerant Computing (FTCS'26)*, Sendai, Japon, pp. 250-259, 1996.

[Snyder 1986] A. Snyder, “Encapsulation and Inheritance in Object-Oriented Programming Languages”, in *OOPSLA'86*, 1986.

[Stein 1987] L. A. Stein, “Delegation is Inheritance”, in *OOPSLA'87*, Orlando, Floride, USA, pp. 138-146, 1987.

[Stroud 1993] R. J. Stroud, “Transparency and Reflection in Distributed Systems”, *ACM Operating Systems Review*, vol. 22(2), pp. 99-103, 1993.

[Stroustrup 1992] B. Stroustrup, *Le Langage C++*, Addison-Wesley, ISBN 2-87908-013-4, 1992.

[Strumpen et Ramkumar 1998] V. Strumpen et B. Ramkumar , “Portable Checkpointing for Heterogeneous Architectures”, in *Fault-Tolerant Parallel and Distributed Systems*, D. Avresky, R. et D. Keli, R., Eds., Kluwer Academic Press, pp. 73-92, 1998.

[Sun 1998] Sun, “Java Object Serialization Specification”, Sun Microsystems, Technical Report, November 1998.

[Tatsubori 1999] M. Tatsubori, “An Extension Mechanism for the Java Language”, Master's thesis, University of Tsukuba, Tsukuba, Ibaraki, Japan, 1999.

[Thibault 1998] S. Thibault, “Langages Dédiés: Conception, Implémentation et Application”, Thèse de Doctorat, Université de Rennes 1, IRISA, Rennes, France, 1998, pp.:127.

[Veríssimo et Marques 1990] P. Veríssimo et J. Marques, “Reliable Broadcast for Fault-Tolerance on Local Computer Networks”, in *IEEE Symposium on Reliable Distributed Systems (SRDS-9)*, pp. 54-63, 1990.

[Visigenic 1997] Visigenic, “Visibroker C++ 3.0 Programmer's Guide”, Visigenic Software Inc, 1997.

[Welch et Stroud 1999] I. Welch et R. Stroud, “From Dalang to Kava: the Evolution of a Reflective Java Extension”, in *Reflection'99*, Saint-Malo, France, pp. 2-21, 1999.

[Yokote 1992] Y. Yokote, “The Apertos Reflective Operating System : The Concept and its Implementation”, in *OOPSLA'92*, New-York, NY, USA, pp. 414-434, 1992.

TABLE DES MATIÈRES

INDEX DES FIGURES

INDEX DES TABLES