Tolérance aux fautes dans les systèmes répartis à base d'intergiciels réflexifs standards

Taha Bennani

N° d'ordre : 780 Année 2005

THÈSE

présentée
pour obtenir le titre de

DOCTEUR DE L'INSTITUT NATIONAL DES SCIENCES APPLIQUEES DE TOULOUSE
Spécialité : Informatique

par

Mohamed Taha BENNANI

Ingénieur ENIT – DEA Systèmes Informatiques

Tolérance aux Fautes dans les Systèmes Répartis à base d'Intergiciels Réflexifs Standards

Soutenue le 20 juin 2005 devant le jury composé de :

MM.	Alain	COSTES	Président
	Pierre	MAURICE	Rapporteur
	Michel	RAYNAL	Rapporteur
	Jean-Paul	BLANQUART	Examinateur
	Juan-Carlos	RUIZ-GARCIA	Examinateur
	Jean-Charles	FABRE	Examinateur

Thèse préparée au Laboratoire d'Analyse et d'Architecture des Systèmes LAAS-CNRS
au sein du Groupe Tolérance aux fautes et Sûreté de Fonctionnement informatique TSF

Thèse de Doctorat de Mohamed Taha BENNANI

« Tolérance aux fautes dans les systèmes répartis à base d'intergiciels réflexifs standards »

RESUME

Conscient que la réflexivité permet d'améliorer la conception et la maintenance des applications, en séparant leurs aspects fonctionnels et non-fonctionnels, nous explorons dans cette thèse le potentiel réflexif de l'intergiciel CORBA. Afin d'effectuer une analyse en profondeur, nous avons développé une plate-forme à composants fournissant de manière transparente des mécanismes classiques de réplication.
Cette plate-forme nous a permis de montrer les limites de la spécification actuelle des intercepteurs CORBA, *PIs* (*Portable Interceptors*). Nous avons identifié précisément certaines difficultés de mise en œuvre, notamment à cause de la dépendance des intercepteurs vis-à-vis du serveur auquel ils sont attachés et la faible contrôlabilité des interactions réparties. À la lumière de ce travail, nous proposons une amélioration du potentiel réflexif de la norme CORBA actuelle afin de la rendre plus adaptée à la mise en œuvre des mécanismes de tolérance aux fautes.

Mots Clefs

- Tolérance aux fautes
- Réflexivité
- Architecture CORBA
- Intercepteurs

"Distributed Systems Fault Tolerance Based on Standard Reflexive Middleware"

ABSTRACT

Reflection makes it possible to improve the design and maintenance of the applications, by separating their functional and non-functional aspects. Based on our analysis of the reflective fault tolerant approaches, we defined a new classification that shows the pertinence of this approach with respect to more conventional ones to provide fault tolerance. The core contribution of this thesis is to explore the reflexive capabilities of the CORBA middleware standard i.e. Portable Interceptors, to build fault tolerant distributed applications. In order to carry out an in-depth analysis of such capabilities, we designed a generic component based platform, called DAISY "*Dependable Adaptative Interceptors and Serialization-based sYstem*", providing replication mechanisms in a transparent way.
DAISY enabled us to show the limits of the current CORBA Portable Interceptors specification. We identified precisely some implementation problems due to some crucial drawbacks of their current definition, in particular, the dependence of the interceptors with respect to the application to which they are attached and the weak controllability of the interactions between them. In the light of this work, we propose an improvement of the current CORBA standard specification in order to enhance its reflective capabilities and adapt the CORBA Portable Interceptors to suit the implementation of fault tolerance mechanisms.

Keywords

- Fault tolerance
- Reflection
- CORBA architecture
- Portable Interceptors

À la mémoire de Chdoula.

À la mémoire de Baba Azizi.

À ma Azaiza qu'aucun mot ne peut décrire ma gratitude.

À mes parents dont le soutien ne ternie jamais.

À mon frère qui aura à me supporter de nouveau.

À ma douce Founa pour sa patience et sa complicité.

À ma petite Meryouma pour la joie qu'elle apporte.

AVANT-PROPOS

Les travaux présentés dans ce mémoire ont été réalisés au Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS), dirigé au cours de mon séjour successivement par Messieurs Jean-Claude Laprie et Malik Ghallab, que je tiens à remercier cordialement pour leur accueil.

J'adresse également mes plus sincères remerciements à Messieurs David Powell et Jean Arlat, directeurs de recherche au CNRS et responsables successifs du groupe Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF) du LAAS-CNRS, pour m'avoir reçu au sein de leur équipe.

J'exprime ma vive reconnaissance à Monsieur Jean-Charles Fabre, Professeur à l'Institut National Polytechnique de Toulouse et Directeur de recherche au CNRS, pour avoir encadré mes travaux de thèse et pour le soutien qu'il m'a accordé.

Je remercie Monsieur Alain Costes, Professeur à l'Institut National Polytechnique de Toulouse, pour l'honneur qu'il me fait en présidant mon Jury de thèse.

Je tiens à remercier également :

- Monsieur Pierre Maurice, Directeur de Recherche à l'INRIA/IRIT,
- Monsieur Michel Raynal, Professeur à l'Université de Rennes,
- Monsieur Juan-Carlos Ruiz-Garcia, Maître de conférences à l'Université Polytechnique de Valence (Espagne),
- Monsieur Jean-Paul Blanquart, Ingénieur d'études à EADS ASTRIUM France,
- Monsieur Jean-Charles Fabre, Professeur à l'Institut National Polytechnique de Toulouse,

pour l'honneur qu'ils me font en participant à mon Jury de thèse, et particulièrement Messieurs Pierre Maurice et Michel Raynal qui ont accepté d'être rapporteurs de mes travaux.

Je remercie particulièrement mon co-encadreur de thèse, M. Marc-Olivier Killijian, chargé de recherche au CNRS, qui a toujours su créer une excellente ambiance de travail fertile en discussions et en réflexions.

Je veux remercier toutes les personnes avec qui j'ai eu l'occasion de collaborer et qui étaient directement liées à mes travaux de recherche : Juan-Carlos Ruiz-Garcia, François Taiani, Eric Marsden, Ludovic Courtes, Benjamin Lussier, Laurent Blain.

Merci à toute l'équipe TSF pour son accueil, sa bonne humeur et son ambiance scientifique très riche. Je veux remercier Joëlle et Gina les deux secrétaires pour leur disponibilité, l'ensemble des permanents pour leur critique et conseil, et l'ensemble des doctorants : Anna "fromasse", Eric "Little bouda", Carlos "Big bouda", Magnus "Salut juyas", Nico "ça va comme un lundi", Arnaud "La faucheuse", Guillaume "Mister Tee", Minh "Chin chao" pour l'ambiance conviviale qu'ils y mettent.

Merci aussi à tous les autres amis que j'ai côtoyé au cours de ces quatre années : Ramy "L'ange syrien", Gourioul "Speedy gonzalez", Céline "Mmm ya des coockies", Philippe "Ahh j'ai mal au ventre, c'est la crème fraîche pourrie", Karim "Badaboum", Mohamed "L'université publique libanaise n'est pas publique!", Najla "tu n'as pas autre chose à dire?", Florent "Mmm ya de la fraise".

Un grand Merci à Slouma "Toutes les marques c'est de la merde, n'achetez que siemens", à Olfa Kaddour "Vous allez voir, je vais gagner", à Allouch "Je ne sais pas pourquoi je ne vais pas bien … kisss…", à Aida "Que la paix soit avec vous, pourriez vous me ramener un sandwich? … svp", à Houda "Ahhhh c'est dégueu, y a de l'oignon!!", à Chohrour "Ne donne pas ton avis, attends pour voir la tendance", à Houda Ben Attia "ya des soldes aux nouvelles galeries", à Jamel "Ya hlili" et à la deuxième Olfa "Tchouka". Vous êtes ma deuxième famille.

Je souhaite remercier tous mes proches : Rwidha "ça te dit d'aller à la ferme demain à 4h00 du matin ?", Fouza "Viens chez moi, Bachra nous prépare une nouvelle spécialité!!", Rchaida "ça te dit un bon Kafteji?", NaNa "Il faut que tu manges bien, tu es entrain de maigrir", Toufa "Mon arrière grand père et la cousine de ton arrière grand mère sont des demi-frères…", Abdou "1, 2 et 3", Mehdouch "J'ai encore fait une charrette cette nuit", Sallouma "5000 Volts", Yousrouna "Na…Na…", Sissouwa "Je connais un coin superbe", oncle Salah "Tu viens avec moi, on ira au marché gabaji", tante Radhia "Laisse le tranquille", Oussaifa "Je vais vous épater avec mon luth", Soumaya "AHH … Allo qui est là …?", Haffoudha "Tu sais, il y a un joueur de foot tunisien à tataouine les bains" et Naoufel "Cool ne t'en fait pas, il n'y a pas le feu" pour leur soutien permanent et infaillible.

TABLE DES MATIERES

CHAPITRE III

MECANISMES REFLEXIFS DANS CORBA ET APPLICATION A LA TOLERANCE AUX FAUTES

CHAPITRE IV

MISE EN OEUVRE DES MECANISMES REFLEXIFS STANDARDS ET ANALYSE

INTRODUCTION GENERALE

Les applications distribuées sont devenues incontournables en termes de déploiement de services, que ce soit dans les systèmes embarqués ou dans les systèmes sur grands réseaux. Afin de répondre aux besoins du marché actuel et de se doter d'un potentiel de réactivité élevé pour satisfaire les nouvelles demandes, les industriels utilisent des environnements d'exécution standards sous forme de composants sur étagères (*Commercial-Off-The-Shelf, COTS*[1]) afin de réduire la complexité et d'écourter la durée de développement de leurs produits, on parle de *time-to-market*.

Du fait de leur situation entre l'application et le système opératoire, ces environnements d'exécution standards sont appelés *intergiciels*. Ils offrent des services de haut niveau, comme l'invocation à distance et la localisation des objets, qui facilitent le développement des applications distribuées en les affranchissant de certains détails de réalisation. Ils utilisent souvent les concepts orientés-objets pour faire abstraction de la complexité du système et promouvoir la modularité et la réutilisation. Ceci permet de disposer de plate-formes particulièrement adaptées à l'intégration des composants hétérogènes.

L'approche à base de COTS permet non seulement la réduction des coûts et des délais de développement mais aussi, l'incorporation rapide des progrès technologiques. Par ailleurs, l'utilisation des composants commerciaux par plusieurs industriels, permet de les considérer comme des standards *de facto*. Ceci se traduit par des avantages sur le plan de la portabilité, de l'évolutivité et de l'interopérabilité. En outre, leur large diffusion permet d'avoir un retour d'expérience pouvant donner des éléments sur leur sûreté de fonctionnement [Arlat 2000].

Si les problèmes liés à la rapidité de mise en œuvre de ces systèmes semblent être maîtrisés, le développement d'applications distribuées sûres basées sur des intergiciels standards COTS relève encore du domaine de la recherche et de l'expérimentation.

Par opposition aux besoins relatifs aux spécifications fonctionnelles de l'application, la gestion de la tolérance aux fautes fait partie des besoins non-fonctionnels du système. En règle générale, la prise en compte de ces besoins contraint les concepteurs à rendre leurs applications plus complexes. Or, l'étroite imbrication des traitements fonctionnels et non-fonctionnels au niveau de l'application est néfaste pour la pérennisation des choix techniques. De plus, devoir intégrer les aspects non-fonctionnels dans les nouveaux programmes d'application est une potentielle source de dysfonctionnement et est également pénalisant vis-à-vis du développement et de la validation des nouvelles applications.

---

[1] L'acronyme Commercial-Off-The-Shelf s'étend à la notion Component-Off-The-Shelf puisque les composants peuvent être d'origine commerciale ou non.

Récemment, la norme CORBA a adopté une extension qui supporte la tolérance aux fautes, à savoir la norme FT-CORBA. Si cette dernière facilite la prise en compte des mécanismes de réplication, elle implique le développeur de l'application dans la mise en œuvre de ces mécanismes non-fonctionnels.

Les travaux présentés dans ce mémoire mettent l'accent sur l'une des techniques permettant la séparation entre les aspects fonctionnel et non-fonctionnel, à savoir la réflexivité. En particulier, nous nous intéressons à l'exploitation du potentiel réflexif de la norme CORBA pour le développement d'une plate-forme à composants assurant les mécanismes de réplication de services à des applications distribuées.

Notre objectif est d'identifier les avantages et les limites du potentiel réflexif de l'intergiciel CORBA pour la mise en œuvre des mécanismes de réplication, aussi bien sur le plan conceptuel que sur le plan de leur implémentation. Enfin, nous proposons une extension de ce potentiel afin de mieux répondre aux besoins de tolérance aux fautes.

Notre mémoire est organisé en quatre chapitres.

Le premier chapitre présente les concepts de base de la tolérance aux fautes dans les systèmes distribués à base d'intergiciels. Il se divise en trois parties : l'introduction de l'intergiciel CORBA et son modèle à objets, puis la présentation des principes de la tolérance aux fautes, à savoir les modèles de fautes et les mécanismes de réplications, et enfin l'introduction de la réflexivité comme moyen de séparation entre les aspects fonctionnels et non-fonctionnels des applications. Nous allons nous appuyer sur ces trois principes pour mettre en œuvre notre étude.

Le second chapitre présente les travaux antérieurs en les classant selon deux grandes approches: les approches classiques et les approches réflexives. Nous définissons dans chaque approche une classification qui permet de dégager les avantages et les inconvénients de chaque solution. Les conclusions de ce chapitre situent notre proposition en analysant les limites des solutions actuelles vis-à-vis de la problématique ciblée.

Le troisième chapitre correspond à la description de la plate-forme DAISY (*Dependable Adaptive Interceptors and Serialization-based sYstem*) que nous avons développée. Il est composé de quatre parties : la présentation des composants de bases standards que nous utilisons pour la mise en œuvre de la réplication, puis la définition de la plate-forme DAISY et de ses différents composants. Deux variantes de cette architecture, basées sur deux formes d'utilisation des intercepteurs standards CORBA, sont ensuite présentées et discutées. Au travers de cette conception, nous identifions les différents problèmes conceptuels (architecturaux) des intercepteurs CORBA. Enfin, nous présentons une proposition d'une nouvelle génération d'intercepteurs, qui est plus adaptée à la mise en œuvre d'architectures tolérant les fautes.

Le quatrième et dernier chapitre présente l'implémentation et l'évaluation de la plate-forme *DAISY* à base de composants externes. Dans une première partie, nous présentons la mise en œuvre de la réplication passive et la réplication active en utilisant les intercepteurs standards CORBA. Au travers de ce développement, nous identifions les différents problèmes pratiques posés par les intercepteurs CORBA. Ensuite, nous proposons une synthèse des avantages et des inconvénients relevés au cours de cette expérimentation et donnons quelques mesures de performances.

CHAPITRE I

INTERGICIELS, TOLERANCE AUX FAUTES ET REFLEXIVITE

Facilitée par l'utilisation de plates-formes de développement dédiées, la distribution des systèmes informatiques représente une solution efficace pour des applications dont les différents acteurs sont localisés dans différents sites géographiquement distants. En effet, cette notion s'est avérée efficace pour des applications bancaires (par exemple, pour mettre à jour des dossiers de clientèles inter-agences), des applications de gestion de production dans différents centres métiers (par exemple, pour concevoir des pièces, dans le cas de l'industrie *AIRBUS*, en coopérant depuis des sites distants), des applications de gestion des dossiers médicaux pour la Sécurité Sociale, etc. Cependant, ces systèmes peuvent présenter des défaillances qui peuvent avoir de sévères conséquences pour leurs utilisateurs ou en terme économique.

Nous présentons dans la première partie de ce chapitre un exemple de plate-forme qui supporte le développement et l'exécution de tels systèmes informatiques distribués, à savoir l'intergiciel CORBA (*Common Object Request Brocker Architecture*). Nous introduisons ensuite différentes notions de sûreté de fonctionnement que nous mettons en œuvre dans ce mémoire, en particulier des modes de défaillances, des modes de communications et des mécanismes de tolérance aux fautes. Une troisième partie présente la réflexivité, qui permet d'améliorer la mise en œuvre de la tolérance aux fautes dans les systèmes distribués. Finalement, nous faisons ressortir la problématique que nous adressons dans nos travaux.

I.1. L'INTERGICIEL CORBA

Un intergiciel est un ensemble de services qui ont pour rôle de faciliter le développement d'applications, qui sont ajoutés au-dessus des bibliothèques systèmes, et qui servent d'intermédiaires entre les applications et le système opératoire. Dans nos travaux, nous nous sommes particulièrement intéressés aux intergiciels orientés communication. Ce type d'intergiciels apporte un ensemble de services qui assurent la communication entre des plates-

formes hétérogènes, facilitant les interactions entre des processus qui s'exécutent sur différentes machines d'un même réseau. En fournissant l'interopérabilité entre différentes plates-formes, les intergiciels ont contribué à la migration des systèmes à architectures centralisées, souvent rigides et contraignantes, vers des systèmes « client/serveur » plus appropriées à la distribution et à la collaboration. Selon la technique qu'ils utilisent, RPC *Remote Procedure Call*, MOM *Message Oriented Middleware* et ORB *Object Request Broker*, les intergiciels présentent des potentiels différents [Youngblood 2004].

Les intergiciels basés sur le concept de l'appel d'une procédure distante (RPC) ont été les premiers logiciels développés, et sont apparus vers le début des années 80 [Birrell 1984]. Ce type d'intergiciels agrège des fonctions spéciales, relatives à l'invocation à distance, au niveau du code de l'application, c'est-à-dire le client et le serveur qui les utilisent. Ainsi, ces intergiciels ne peuvent pas exister seuls et leur portabilité est limitée. Par ailleurs, ces intergiciels ne sont pas appropriés pour le développement des applications orientées objets.

Les intergiciels orientés messages (MOM) qui ont paru vers la fin des années 80 [Steinke 1995], sont adaptés aussi bien aux applications procédurales qu'aux applications à objets. Ils ne sont pas liés à l'application mais se présentent sous la forme d'une couche logicielle indépendante. Toutefois ces intergiciels restent des solutions propriétaires (c'est-à-dire non standard) puisqu'ils ne spécifient pas une architecture commune. Ceci limite l'interopérabilité des applications qui utilisent des intergiciels issus de différents fournisseurs.

Le troisième type d'intergiciels est basé sur la notion de bus à objets (ORB). Apparu au début des années 90 [Wade 1993], le bus à objets sert de support aux applications en masquant une partie de la complexité de la programmation distribuée à objets tout en mettant l'accent sur l'interopérabilité des applications. Ainsi, différentes applications peuvent communiquer entre elles en utilisant un ORB. Plusieurs intergiciels de ce type ont été développés, tels que COM *Component Object Model* et DCOM *Distributed Component Object Model* de Microsoft [Williams 1994], RMI *Remote méthode Invocation* de JAVA [Sun 2004] ou CORBA de l'OMG (*Object Management Group*). Dans ce mémoire, nous nous intéressons en particulier à la norme CORBA, qui est la plus répandue pour supporter des applications à objets distribuées.

L'intergiciel CORBA (*Common Object Request Broker Architecture*) [OMG 2002a] est une infrastructure spécifiée par l'OMG; ainsi que nous l'avons indiqué, il est basé sur la notion de l'ORB. L'OMG est un consortium essentiellement composé d'acteurs de l'industrie du logiciel créé en 1989 pour promouvoir l'adoption de standards pour le développement d'applications à objets distribués. Plus précisément, l'objectif de l'OMG est de simplifier le développement en fournissant un environnement qui, d'une part, masque les détails de l'utilisation des services offerts par les systèmes opératoires, et d'autre part, qui soit supporté par les différents systèmes opératoires actuels et à venir. D'après ces principes, la programmation distribuée deviendrait standardisée et inter-opérable, au sens où elle serait indépendante des exécutifs, par exemple du système opératoire et de la couche matérielle de la machine hôte, et des langages de programmation (cf. Figure 1).

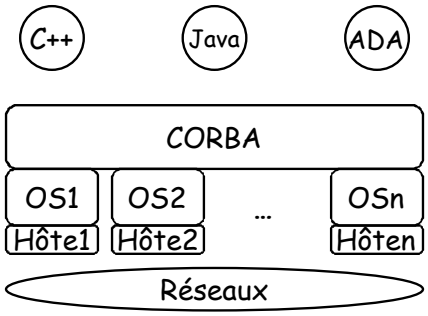


Figure 1. Vue globale de l'intergiciel CORBA

I.1.1. Le modèle à objet de l'intergiciel CORBA

Dans cette partie, nous présentons les définitions des entités qui sont utilisées par le modèle à objet CORBA :

- Un *client* est une entité qui, conceptuellement, peut ne pas être un objet. Un client requiert un service auprès d'autres objets.

- Un *objet* est une entité identifiable, qui fournit un ou plusieurs services pour d'éventuels clients. Un objet est caractérisé par un état, un comportement et une identité. L'état d'un objet renferme la description de toutes ses fonctionnalités et les valeurs courantes de ses attributs. Le comportement précise la manière avec laquelle un objet agit et réagit en fonction du changement de son état et des messages reçus qui déclenchent des opérations internes. L'identité est la propriété qui permet de distinguer deux objets, qu'ils aient ou non le même type. Le processus de création d'un objet s'appelle instanciation.

- Une *interface* renferme l'ensemble des opérations qu'un client peut invoquer auprès d'un objet. Toute interface est décrite dans un langage de description d'interfaces, appelé *Interface Definition Language* (IDL). Un objet n'est capable de satisfaire la définition d'une interface que s'il implémente toutes les méthodes spécifiées et déclarées dans celle-ci.

- Une *opération* représente un service offert par un serveur à travers son interface. Elle est décrite dans le langage IDL qui permet de spécifier sa signature, c'est-à-dire le type de chacun de ses paramètres. Les opérations d'un serveur, dans le concept à objet CORBA, correspondent aux méthodes publiques de l'objet qui implémente l'interface de ce serveur, dans les langages orientés-objets.

- Une *requête* est un message issu d'un client pour effectuer une demande de service auprès d'un objet offrant un rôle de serveur. L'intergiciel interprète ce message pour décider du serveur qu'il va appeler et du service que ce dernier va effectuer. Cette interprétation consiste à extraire de la requête l'identificateur du serveur cible et le nom de l'opération à invoquer. Le premier sert à identifier d'une manière unique un serveur parmi ceux qui sont présents dans le réseau et le second permet de repérer une

opération parmi l'ensemble de celles qui sont définies au niveau de l'interface du serveur. Par ailleurs, la requête renferme les paramètres de l'opération qui véhiculent les données d'entrée du service.

- Un *attribut* est déclaré au niveau de l'interface d'un serveur. Bien qu'il corresponde à la notion d'attribut d'une classe dans le concept orienté-objet standard, il ne peut être que public. Pour qu'un client distant puisse récupérer ou changer la valeur d'un attribut, l'intergiciel lui associe une paire d'opération appelée *accesseurs*.

I.1.2. L'architecture de l'intergiciel CORBA

L'architecture de l'intergiciel CORBA est présentée sur la Figure 2.

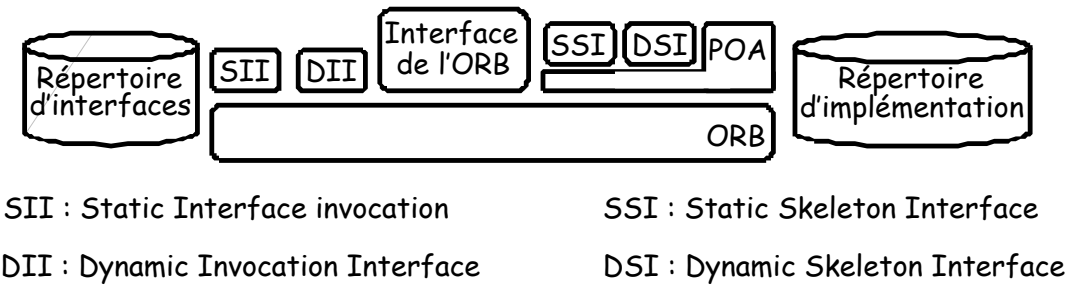


Figure 2. Architecture de l'intergiciel CORBA

Cette architecture est organisée autour de la notion d'ORB (*Object Request Brocker*) qui correspond à un bus à objets responsable de la connexion et de l'interaction entre objets. Cette architecture requiert également les services suivants :

- Le *répertoire d'interfaces* (Interface Repository) est un référentiel qui stocke la description des interfaces sous la forme d'objets accessibles par le client et l'ORB (Object Request Brocker). L'objectif est d'accéder lors de l'exécution à une représentation objet des interfaces des objets présents sur le bus.

- L'*interface d'invocation statique* (Static Invocation Interface, SII) appelée aussi « souche », est le résultat de la pré-compilation d'une interface IDL dans un langage donné selon une projection donnée, c'est-à-dire dans un langage de programmation donné. Du point de vue du client, l'utilisation des souches s'apparente à des appels locaux puisqu'elles jouent le rôle de médiateurs (*proxies*) locaux d'objets-serveurs qui résident dans les sites distants.

- L'*interface d'invocation dynamique* (Dynamic Invocation Interface, DII), est une interface unique et indépendante de celle du serveur cible. Elle offre aux clients la possibilité d'interagir avec des serveurs dont l'interface est inconnue lors de leur génération. Cette API (*Application Programming Interface*) normalisée par CORBA permet de lire l'interface d'un serveur, de générer des paramètres, de créer des requêtes, de les invoquer dynamiquement et de récupérer les résultats.

- L'*interface de l'ORB* est celle qui donne accès aux primitives de base de l'ORB (par exemple, la conversion de référence objet/chaîne de caractères, l'initialisation de l'ORB, …). Elle est commune aux applications clients et serveurs.

- L'*interface statique du squelette* (Static Skeleton Interface, SSI) fournit des interfaces pour les invocations statiques des objets serveurs ; une SII est nécessaire par objet. Elle décode les requêtes pour invoquer le code des méthodes appropriées. Comme pour les souches, les squelettes sont le résultat de la pré-compilation de l'interface IDL du serveur.

- *L'interface dynamique du squelette* (Dynamic Skeleton Interface, DSI) fournit des mécanismes de lien dynamique à l'exécution pour les objets qui veulent prendre en compte les requêtes adressées à des interfaces non connues lors de leur génération. Ces mécanismes analysent les paramètres de la requête pour déterminer l'objet cible et la méthode à appeler. Ainsi, la DSI côté serveur est équivalente à la DII côté client.

- *L'adaptateur d'objet portable* (Portable Object Adapter, POA) permet aux développeurs d'avoir une implémentation d'objet qui est portable entre les différents ORB. Selon la politique de création choisie, le POA peut générer un ou plusieurs identificateurs d'objet lors de la création d'un serveur. Ces identificateurs sont utilisés par le POA pour faire parvenir de manière transparente une requête au serveur cible. Le POA peut également masquer le redémarrage d'un serveur par rapport à un client en fournissant des identités persistantes aux objets.

- *Le répertoire d'implémentation* (Implementation Repository) est un référentiel d'implémentation qui stocke les détails spécifiques à l'ORB concernant l'implémentation des serveurs, leurs politiques d'invocation ainsi que leurs identités.

Pour envoyer une requête, le client peut utiliser l'interface des invocations dynamiques ou la souche, selon que l'interface du serveur cible est connue ou non lors de son développement. Par ailleurs, le client peut utiliser l'interface de l'ORB pour réaliser certaines fonctions, comme l'initialisation de l'ORB ou la récupération de la référence d'un objet distant.

I.1.3. Dynamique de l'intergiciel CORBA

Nous utilisons dans cette section une application élémentaire pour illustrer d'une part les différentes étapes d'élaboration d'un service, et d'autre part, les transformations que subit la requête issue d'un client avant d'arriver au serveur. L'exemple que nous avons choisi est celui d'un serveur qui affiche le message « Hello World » lorsqu'un client invoque la méthode `say_hello` de l'interface du serveur.

I.1.3.1. Mise en œuvre d'une application

Comme le montre la Figure 3, la création d'un serveur `Hello` nécessite quatre étapes différentes :

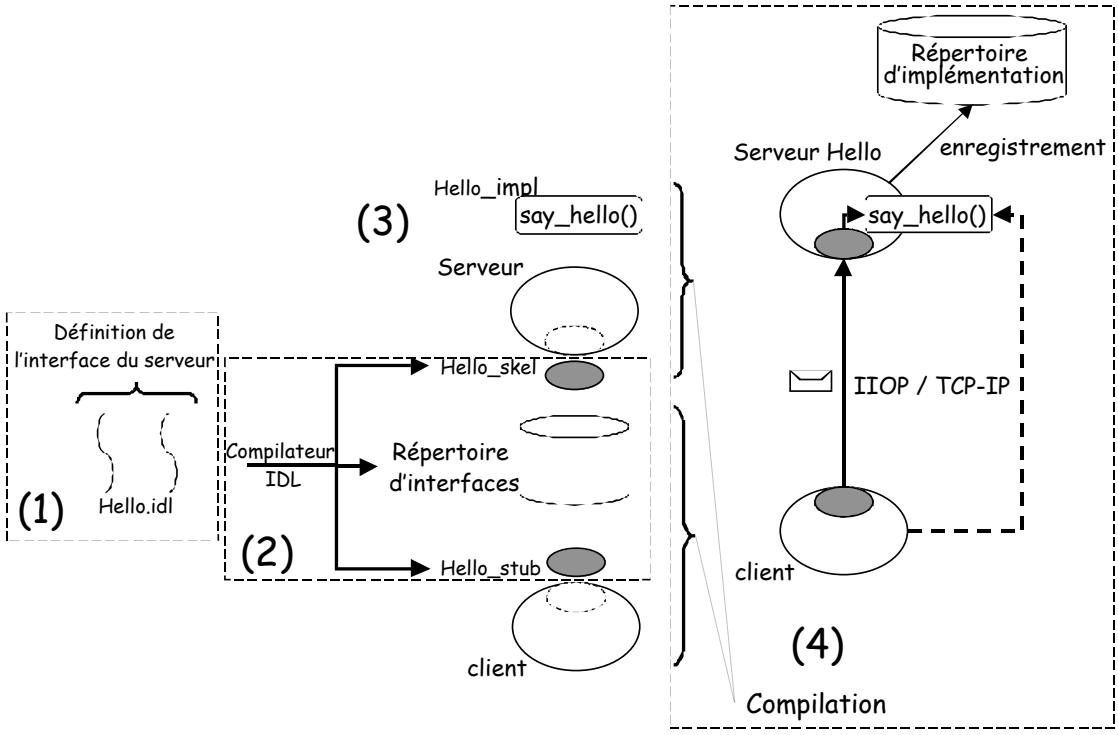


Figure 3. Etapes du développement d'une application CORBA

(1) La définition de l'interface du serveur. Dans cette première étape, le développeur de l'application est amené à spécifier le service que l'objet offrira, en fournissant une interface écrite en IDL (`Hello.idl`). Outre la définition de l'interface de l'objet, ce fichier peut renfermer une hiérarchie d'interfaces ainsi que la définition de nouveaux types nécessaires à la définition de la signature d'une opération.

(2) La seconde étape est automatique. Elle consiste à générer un squelette (`Hello_skel`), une souche (`Hello_stub`) et un répertoire d'interfaces par interface d'objet. Le squelette a pour rôle de décoder les requêtes qui sont arrivées au niveau du serveur pour invoquer l'opération appropriée. Du côté client, la souche offre une interface identique à celle du serveur : ainsi, au lieu d'invoquer l'opération du serveur à distance, le client appelle la méthode de la souche qui représente le service qu'il demande auprès de l'objet serveur. C'est la souche qui s'occupe de l'élaboration de la requête à destination du serveur cible (marshalling). Quant au répertoire d'interface, il joue le rôle d'une base de données qui renferme la description des interfaces et qui est accessible aux clients lors de l'exécution.

(3) La troisième étape est assurée par le développeur de l'application, et consiste à mettre en œuvre le service qu'un objet va rendre. Ceci se traduit par le développement des opérations du côté serveur. Le développeur de l'application doit également implémenter le client qui demandera les services fournis par le serveur.

(4) La quatrième et dernière partie est automatique. Elle consiste en la compilation de l'application et l'intégration du support d'acheminement des requêtes (souches et squelettes) à l'application (client et serveur). Par ailleurs, à l'installation d'un serveur, les informations relatives à l'implémentation de l'objet (c'est-à-dire les informations relatives à l'allocation de

ressources, à la sécurité, à la politique d'activation, à l'exécution des objets et au déverminage) sont enregistrées dans le répertoire d'implémentation. Il permet à l'ORB de localiser et d'activer l'implémentation d'un objet.

Ces applications peuvent être réalisées dans différents langages de programmation, et déployées sur différents types de systèmes opératoires où la représentation des données n'est pas forcément comparable, ce qui pose des problèmes d'interopérabilité. Pour répondre à cette difficulté, la norme CORBA a défini un ensemble de protocoles qui vont de l'identification d'un objet distant au format des données échangées entre différents ORB : l'IOR, le GIOP et l'IIOP qui sont présentés ci-après.

IOR (*Interoperable Object Reference*)

C'est un format standard pour coder la référence d'un objet CORBA. Il contient un en-tête qui représente l'identificateur du répertoire d'interface et un ou plusieurs *Profiles* décrivant où et comment on peut contacter cet objet (c.f. Figure 4).

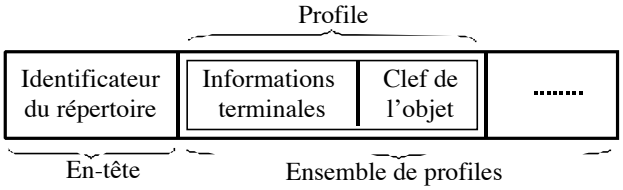


Figure 4. Contenu d'une référence d'un Objet CORBA, [Henning 1999]

L'identificateur du répertoire d'interface est une chaîne de caractères qui identifie d'une manière unique l'interface IDL de l'objet. Les informations terminales (*Endpoint Info*) permettent à l'ORB d'établir une connexion physique avec le serveur implémentant l'objet. Ils indiquent le protocole de communication à utiliser et contiennent les informations relatives à l'adresse physique du serveur cible. Contrairement aux deux premiers champs, la clef de l'objet (*Object Key*) ne possède pas un format standard. Ce champ est destiné à mettre des informations propriétaires. Cependant, la majorité des ORB l'utilisent pour y mettre un identificateur spécifique lors de la création de la référence de l'objet qui est utilisée par le POA pour identifier le serveur cible à chaque arrivée de requête.

GIOP (*General Inter-ORB Protocol*)

C'est un protocole qui permet de mettre les messages qui transitent à travers l'ORB sous un format interoperable et indépendant du protocole de communication sous-jacent. Il définit, pour cela, un ensemble de messages qui sont utilisés, d'une part, par les clients pour envoyer leurs requêtes (i.e. *Request*, *LocateRequest*, *CancelRequest*, *Fragment* et *MessageError*) et, d'autre part, par les serveurs pour retourner leurs réponses (i.e. *Reply*, *LocateReply*, *CloseConnection*, *fragment* et *MessageError*).

La Figure 5 représente le contenu d'un message GIOP.

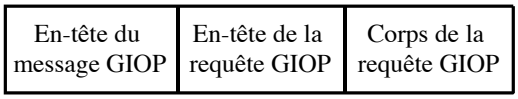


Figure 5. Contenu d'un message GIOP encodant une requête, [Henning 1999]

Ce message est composé de :

- L'en-tête du message GIOP qui spécifie la version du protocole, le type de codage de l'information (c'est-à-dire l'emplacement du bit de poids le plus fort, en d'autres termes *Big endian* ou *little endian*) et le type du message encodé (par exemple Request, LocateRequest, etc.).

- L'en-tête de la requête GIOP qui renferme le nom de l'opération à invoquer, l'identificateur de la requête et la clef de l'objet cible.

- Le corps de la requête GIOP qui renferme les paramètres *in* et *inout* pour un message de type requête, et les paramètres *out* et *inout* pour les messages de type réponse. Les paramètres *in* fournissent les données d'entrée au service, les paramètres *out* peuvent renfermer le résultat d'un service et les paramètres *inout* jouent le double rôle.

IIOP (*Internet Inter-ORB Protocol*)

Ce protocole spécifie comment un message GIOP est *mappé* afin d'être transmis en utilisant le protocole de communication TCP/IP (*Transmission Control Protocol/Internet Protocol*). Ainsi, le protocole IIOP est une implémentation de GIOP sur TCP/IP. Son rôle par rapport à l'implémentation du GIOP est similaire à celui d'un langage de programmation par rapport à l'implémentation d'une interface IDL, c'est-à-dire il transforme le message GIOP en un message IIOP lorsque le TCP/IP est utilisé comme protocole de communication.

I.1.3.2. Interactions entre client et serveur

La Figure 6 montre le chemin d'une requête issue du client à destination du serveur `Hello`.

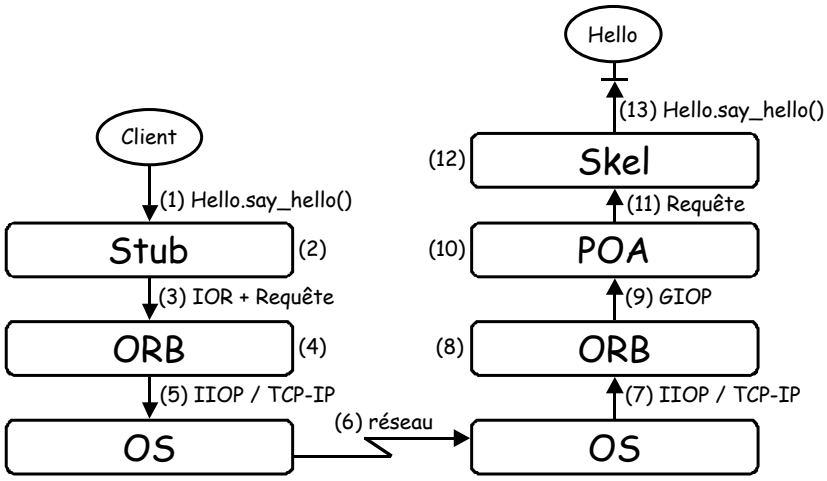


Figure 6. Transformations d'une requête

L'invocation de l'opération `say_hello()` de l'interface du serveur `Hello`, qui est potentiellement distant, est traduite par une invocation locale de la même opération mais au niveau de l'interface de la souche *Stub* (1). Le rôle de la souche est de traduire les paramètres de l'invocation, émise par le client, dans un format compréhensible par différents serveurs. Cette mise en forme de l'invocation en une suite d'octets selon le format CDR (*Common Data Representation*) est appelée empaquetage des paramètres *marshalling* (2). La séquence d'octets représentant les paramètres de l'invocation est ajoutée au nom de l'opération, à l'identificateur de la requête et à l'identificateur de l'objet distant que l'on extrait de l'IOR pour former le message GIOP (3). Ce message est indépendant du réseau de communication sous-jacent (par exemple : TCP/IP, UDP ou IPX). Le message GIOP est ensuite transformé au niveau de l'ORB, selon le profil extrait de l'IOR, en un message compréhensible par la couche de transport, par exemple le format IIOP pour une couche de transport TCP/IP (4). Juste avant l'émission du message, l'ORB gère l'initialisation de la connexion avec le serveur distant. Une fois la connexion établie, et dans le cas d'un réseau TCP/IP, le message IIOP est envoyé par l'ORB sous la forme de paquets IP (5). Ces paquets sont envoyés à leur tour par le système opératoire et vont transiter via le réseau jusqu'à ce qu'ils arrivent au port physique de la machine cible (6). Le processus système relatif au serveur est à l'écoute des requêtes qui arrivent au niveau de son port et envoie les messages récupérés à l'objet en question. En effet, dès que les paquets IP arrivent au niveau du port de communication de la machine hôte, le processus système du serveur les traite et les délivre à l'ORB (7). Une première transformation est apportée par l'ORB en générant un message IIOP (8). Une seconde transformation, à la sortie de l'ORB, est apportée au message IIOP pour le présenter sous une forme standard *GIOP* compréhensible par le POA cible (9). Il est à noter que le choix du POA est réalisé à partir des informations du champ *Object key* du message. Le POA utilise aussi le champ *Object key* du message GIOP pour déterminer le squelette *skeleton* du serveur qui va traiter la requête (10). Une fois arrivé au niveau du skeleton (11), ce dernier localise l'opération associée, dépaquète le corps du message en paramètres pour l'appel de l'opération *demarshalling* (12), pour terminer par l'appel à l'opération de l'interface IDL du serveur *operation upwelling* (13).

Nous avons pu ainsi voir qu'une application distribuée, utilisant l'intergiciel CORBA et fournissant un service particulier, est composée :

- du code de l'application,
- de librairies et de services fournis par l'intergiciel sous forme binaire, et
- de souches et de squelettes qui sont générés automatiquement par des compilateurs et qui doivent être compilés et liés aux deux premières composantes.

Nous avons présenté dans cette partie les différents concepts liés à l'intergiciel CORBA (à savoir le modèle à objet et la terminologie utilisée), son l'architecture et sa dynamique. Nous utilisons la terminologie du modèle à objet CORBA pour présenter nos travaux ainsi que les travaux connexes, qui s'articulent autour des applications distribuées. La compréhension des éléments de bases de l'architecture CORBA sera utile, dans la suite de ce manuscrit, pour comprendre l'architecture que nous y proposons. Dans la partie présentant la dynamique de l'intergiciel nous illustrons le comportement de l'intergiciel CORBA sans mécanismes de tolérance aux fautes. L'aspect relatif à la tolérance aux fautes dans les systèmes à base d'intergiciels sera traité dans le chapitre suivant. Dans la section qui suit, nous présentons les principes de la tolérance aux fautes ainsi que les techniques utilisées pour construire un système distribué sûr de fonctionnement.

I.2. LA TOLERANCE AUX FAUTES

I.2.1. Définitions

La sûreté de fonctionnemen*t d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. 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 interagit avec le système considéré* [Laprie 1995].

Les propriétés d'un système sûr de fonctionnement sont souvent corrélées à son domaine d'application. Ainsi, un système de contrôle-commande d'un avion est dit sûr de fonctionnement, si et seulement s'il est, d'une part disponible (c'est-à-dire toujours prêt à être utilisé) et, d'autre part, fiable (c'est-à-dire qu'il délivre son service de façon continue sur une période temporelle donnée). D'autres **attributs** comme la confidentialité ou l'intégrité peuvent être plus appropriés pour qualifier d'autres systèmes sûrs de fonctionnement, comme les applications bancaires ou du domaine de la santé publique. Dans ce mémoire, nous nous focalisons sur les notions de disponibilité et de fiabilité.

Lors du développement d'un système informatique (distribué ou non), des ***fautes*** peuvent être introduites au cours de sa conception et/ou de son implémentation sans être corrigées lors de

la phase de validation. Les ***erreurs*** correspondantes restent latentes pendant la durée de vie du système jusqu'à ce qu'elles soient activées. Par exemple, dans le système de gestion d'archivage du véhicule martien SPIRIT, l'oubli de la libération de l'espace mémoire après l'envoi des images sur la terre qui représente une erreur latente n'a été activée que lorsque le système d'acquisition avait saturé la zone de stockage des données : ceci a conduit à deux semaines d'indisponibilité. L'origine d'une ***erreur*** est une faute, qui dans la plus part des cas est d'ordre conceptuel ou physique. En d'autres termes, on peut souvent ramener l'origine d'une erreur à une faute humaine au niveau de la conception d'un système ou d'une faute physique d'un composant matériel, et ce, de façon causale. L'activation d'une erreur d'un composant d'un système distribué peut affecter le service qu'il rend, auquel cas le composant est déclaré ***défaillant***. Par ailleurs, la défaillance d'un composant représente une ***faute*** auprès des autres composants avec lesquels il interagit. Les défaillances, les erreurs et les fautes représentent les **entraves** à la tolérance aux fautes.

La tolérance aux fautes représente la capacité d'un système à remplir sa fonction en dépit des fautes [Avizienis 2004]. D'autres **moyens** de la sûreté de fonctionnement peuvent être envisagés pour bâtir des systèmes sûrs de fonctionnement (i.e. l'élimination, la prévention ou encore la prévision des fautes). Dans le cadre du travail présenté dans ce manuscrit, nous nous intéressons exclusivement à la tolérance aux fautes.

I.2.2. Modèles

Pour pouvoir concevoir des systèmes distribués tolérant aux fautes, nous sommes amené à modéliser, d'une part, la communication entre leurs différents composants, et d'autre part, leurs modes de défaillance. Si ce deuxième modèle ne dépend que des hypothèses de fautes dans les composants, le premier ne prend pas en considération seulement le modèle de communication défini au niveau des composants mais également les hypothèses de synchronisme des communications. Ainsi, la définition de ces différents modèles permettra de définir la technique adéquate pour la tolérance aux fautes.

I.2.2.1. Modes de défaillances

Pour un utilisateur unique, un système peut **défaillir en valeur** ou **temporellement**. En effet, si la valeur du service délivré ne permet plus l'accomplissement de la fonction du système, ce dernier est déclaré défaillant *par valeur*. D'autre part, si les conditions de délivrance du service ne permettent plus l'accomplissement de la fonction du système, soit d'une façon permanente (arrêt) ou de façon transitoire (dépassement d'échéance), celui si est considéré comme défaillant *temporellement*.

En revanche, dans le cas où plusieurs utilisateurs pourraient demander la même fonction auprès d'un service, la défaillance d'un composant peut être perçue comme :

- **Cohérente** : tous les utilisateurs du systèmes ont la même perception des défaillances.

- **Byzantine** (incohérente) : les utilisateurs du système peuvent avoir des perceptions différentes des défaillances [Lamport 1982].

Ainsi, pour mettre en œuvre des mécanismes de tolérance aux fautes efficaces, il est impératif d'avoir une définition formelle des modes de défaillances. Un raffinement de la classification des modes de défaillance proposée dans [Powell 1992] consiste à décomposer les modes de défaillance en modes d'erreurs élémentaires définis formellement. Toute défaillance est repérée par un couple d'assertions **(V,T)**, où **V** représente l'assertion relative à l'erreur en **valeur** et **T** celle relative au **temps**. On distingue alors plusieurs modes de défaillances : le silence sur défaillance, la défaillance consistante et la défaillance incontrôlée. Dans un but de clarté et de concision, nous nous limitons dans ce qui suit à la présentation des deux modes que nous utiliserons dans les chapitres suivants :

- *Système à silence sur défaillance* : c'est un système « parfait » puisqu'il ne produit pas d'erreurs. Qualifié aussi de système auto-testable, il est repéré par le couple $\mathbf{(V_{sd},T_{sd})}$.

$\mathbf{V_{sd}}$ **:** Tout message envoyé par le système est correct. C'est-à-dire, il appartient à un ensemble de valeurs connu à l'avance et il est identique pour tous les clients.

$\mathbf{T_{sd}}$ **:** Aucun message n'est délivré après l'occurrence d'une erreur. En revanche, lorsque le système est correct, le message est délivré au bout d'une période de temps bornée et connue à l'avance.

- *Système à défaillance incontrôlée* : c'est un système plus réaliste que celui qui est à silence sur défaillance. Il considère qu'un système peut délivrer une valeur erronée et au bout d'une période non connue. Ce mode de défaillance est repéré par le couple $\mathbf{(V_{du},T_{du})}$.

$\mathbf{V_{du}}$ **:** Un message envoyé par le système peut avoir une valeur correcte ou non. Par ailleurs, ce système peut délivrer des messages différents pour une même requête.

$\mathbf{T_{du}}$ **:** Le système peut produire un message au bout d'un intervalle de temps borné et connu à l'avance, comme il peut le produire avec un retard. Par ailleurs, le temps séparant deux messages différents n'est pas borné.

Cette classification permet non seulement de définir formellement les modes de défaillances, mais aussi de les ordonner d'une manière partielle. En effet, un mécanisme de tolérance aux fautes dédié à des erreurs vérifiant une première assertion, devra pallier les erreurs vérifiant une seconde assertion si les erreurs prises en compte par la première couvrent un spectre plus large. Par exemple, les mécanismes de tolérance aux fautes élaborés pour un système à défaillance incontrôlées seront applicables à un système à silence sur défaillance.

I.2.2.2. Modèles temporels

Même si le mode de défaillance d'un composant est défini, la mise en place de mécanismes de tolérance aux fautes dans les systèmes distribués doit prendre en compte d'une

part, le temps que nécessite l'exécution d'une fonction et d'autres part, le délai introduit par la communication entre les différents composants du système. Cette caractérisation de la durée de délivrance d'une réponse, relève du modèle temporel du système distribué.

Il est possible de classifier un système selon les trois modèles temporels suivants :

- **Modèle asynchrone** : Appelé aussi modèle sans temps, il est le modèle le plus réaliste puisqu'on ne fait aucune hypothèse sur les temps d'exécution et de délivrance 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, sans pouvoir garantir de borne temporelle [Powell 1998]. Le choix de ce modèle ne permettant de faire aucune hypothèse temporelle, rend la résolution de certains problèmes très difficile, voire impossible à résoudre. En effet, dans ce type de systèmes, où un nœud lent ne peut être distingué d'un nœud défaillant, les problèmes fondamentaux de la tolérance aux fautes, comme le problème du consensus, ne peuvent pas être résolus d'une manière déterministe [Fischer 1985]. En revanche, l'intérêt que sollicite ce modèle temporel réside dans le fait qu'il ne pose pas de contraintes et peut par conséquent s'appliquer à tous les systèmes. Ceci facilite la focalisation sur les propriétés fondamentales du système sans complication de son analyse avec des considérations d'ordre pratique. Ainsi, si un problème est résolu dans ce modèle, il le sera aussi pour un système réel [Birman 1996].

- **Modèle synchrone** : A l'autre extrémité du spectre des modèles temporels, le modèle synchrone, appelé aussi modèle à délais bornés à priori, est le plus rigide. Il suppose les propriétés suivantes[2] :

  1)- Les horloges des différents nœuds sont à dérive bornée, c'est-à-dire la vitesse relative de ces horloges est bornée par rapport à celle de l'horloge de référence,

  2)- Les délais de communication et de traitement sont bornés. En particulier, tout message envoyé par un processus non défaillant à un autre processus est reçu et traité dans un intervalle de temps borné.

  En pratique, pour remplir de telles exigences, il est nécessaire, d'une part, d'utiliser des techniques d'ordonnancement temps réel, et d'autre part, de supposer que le nombre de défaillances pouvant affecter le système par intervalle de temps donné est borné [Fetzer 1997]. Ce modèle est particulièrement adapté aux applications critiques avec des contraintes temporelles strictes même en présence de faute. Cependant, il s'appuie sur des hypothèses dont l'évaluation de la couverture exigerait une connaissance parfaite du système d'exploitation, du système de communication et de leur charge de travail. L'intérêt d'un tel modèle est essentiellement d'établir une borne inférieure aux

[2] Certains auteurs supposent aussi l'existence d'un temps global. De façon générale, on s'appuie sur le fait que les délais de communications sont bornés pour construire un temps global.

solutions proposées pour les problèmes d'algorithmique distribuée. En effet, si on prouve qu'un problème n'a pas de solution sous l'hypothèse du synchronisme, alors il n'aura pas de solution sous d'autres hypothèses [Birman 1996].

- **Modèle asynchrone temporisé** : Entre le modèle asynchrone, où aucune hypothèse temporelle n'est faite, et le modèle synchrone, où tous les délais sont parfaitement connus, se trouve le modèle asynchrone temporisé plus réaliste. Dans ce modèle, on ne fait pas d'hypothèses fortes sur les aspects temporels et l'on admet la possibilité que certaines hypothèses ne puissent pas être satisfaites à un moment donné. En particulier, ce modèle suppose que les délais de transfert et de traitement des messages peuvent éventuellement être bornés à posteriori par les utilisateurs et que les horloges ne sont pas synchronisées mais ont une dérive bornée et connue par rapport au temps réel. La borne « $\Delta = s + d + s$ » [Cristian 1996] est calculée sur la base des retards liés :

  - à l'ordonnancement de la tâche d'envoi (côté producteur) et de réception (côté consommateur) par le processeur. Cette constante est choisie telle que la plupart des processus ait un temps d'exécution inférieur ou égal à *s*.

  - au délai de transmission du message par le canal. Cette constante est choisie telle que la majorité des messages ait un délai de transfert inférieur ou égal à *d*.

  Dans ce modèle, tout message qui subit un retard plus élevé que ce qui est permis est dit non performant. Le modèle asynchrone temporisé s'applique aisément à la plupart des systèmes distribués existants. Ainsi, plusieurs travaux définissant des mécanismes de tolérance aux fautes s'inscrivant dans ce modèle ont été adoptés dans des applications réelles, en particulier dans celles qui s'inscrivent dans le cadre des systèmes à défaillances incontrôlées.

Dans le cadre de ce mémoire, nous utilisons le modèle asynchrone temporisé en dépit de l'utilisation de l'intergiciel CORBA dont la communication est du type asynchrone. L'utilisation des temporisateurs nous permettra cependant de détecter les défaillances au prix de certains faux positifs et d'un délai de détection relativement long.

I.2.3. Mise en œuvre de la tolérance aux fautes

Le choix d'un modèle de faute et d'un modèle temporel permet de déterminer le nombre de répliques nécessaires pour fournir un service correct en présence de fautes en fonction du niveau désiré de tolérance aux fautes. Par exemple, un système à silence sur défaillance peut tolérer K fautes, s'il est répliqué en K+1 serveurs. La mise en œuvre de la réplication requiert généralement l'utilisation de certains services comme un détecteur de défaillances et un système de communication de groupe.

Nous présentons ci-après les différentes stratégies de réplication et les différents rôles joués par un système de communication de groupe.

I.2.3.1. Réplication

L'idée d'utiliser la redondance dans les systèmes informatiques pour masquer les défaillances des composants a été introduite par Von Neumann [Neumann 1956]. Avec plusieurs répliques, une entité répliquée continue à fournir un service à un client même si une ou plusieurs répliques sont défaillantes. Dans un système informatique, la redondance peut être utilisée au niveau du stockage des données, des ressources de calcul, des liens de communication entre client et serveur, ou encore au niveau des composants de l'application elle même.

Dans les systèmes distribués, trois modes principaux de réplication sont envisageables : la réplication active, semi-active ou passive. Le choix entre ces différents mécanismes se fait selon les hypothèses de fautes et le domaine d'application.

- **Réplication active (*N-modules replication*)** : dans ce protocole de réplication, chaque réplique joue le même rôle : chacune reçoit la requête émise par le client, la traite, met à jour son état interne puis retourne sa réponse au client émetteur. Dans ce cas, le protocole de réplication n'est pas centralisé puisque chaque réplique fournit les réponses. Par ailleurs, et comme les requêtes sont envoyées à toutes les répliques, la défaillance de l'une d'entre elles est transparente du point de vue du client. En effet, si on prend comme hypothèses que (H1) toutes les répliques sont déterministes, que (H2) le canal de communication est non-défaillant, que (H3) les requêtes arrivent et sont traitées par les serveurs dans le même ordre et que (H4) ces derniers ne défaillent que sur arrêt, la première réponse qui arrive au client sera considérée comme correcte. Ceci masquerait, en cas de retour de réponse, la défaillance probable d'au plus N-1 serveurs, N étant le nombre total des répliques. Par ailleurs, si l'hypothèse (H4) est relaxée (i.e. un serveur ne peut retourner une valeur erronée), un vote sur les réponses retournées sera réalisé et c'est la valeur issue du vote qui sera délivrée au client. Dans ce cas, on aura besoin de 2N+1 répliques pour pouvoir tolérer N erreurs par valeurs simultanées.

- **Réplication passive (*primary-backup replication*)** : à l'inverse de la réplication active, ce protocole est centralisé au niveau du serveur **primaire** ; les autres répliques appelées aussi **secondaires** (*backups*) ne reçoivent pas la requête du client. En effet, dans cette stratégie, un client envoie une requête au primaire uniquement, celui-ci l'exécute, met à jour l'état des secondaires à partir de son propre état, puis retourne la réponse au client. Si le primaire défaille (sur arrêt), un serveur parmi les répliques secondaires prend la relève. Par ailleurs, en utilisant la réplication passive, nous n'avons plus besoin de l'hypothèse de l'atomicité (H3), puisqu'elle est intrinsèque d'une part à la linéarisation des messages reçus et envoyés par le primaire, et d'autre part, à l'attente de l'acquittement envoyé par les secondaires et attendu par le serveur primaire. L'hypothèse du déterminisme (H1) n'est pas non plus nécessaire, puisque seul le serveur primaire traite la requête, ce qui distingue cette approche de la réplication active en termes d'utilisation des ressources. Ainsi, un tel mécanisme de réplication est le plus rapide pour le traitement des requêtes en l'absence de fautes. En

revanche, cette approche est plus lente pour retourner une réponse dans le cas d'une défaillance du primaire. En plus, elle est plus difficile à mettre en œuvre puisque l'application qui l'utilise doit fournir des mécanismes de mise à jour de l'état (capture et envoi). Cette approche est très utile pour les applications sans changement d'état comme dans le cas de la consultation et non pas de mise à jour des bases de données.

- **Réplication semi-active (*leader-follower replication*)** : à mi-chemin entre la réplication active et la réplication passive, cette approche définit un meneur (*leader*) et plusieurs suiveurs (*followers*) qui traitent les requêtes envoyées par le client (comme dans le cas de la réplication active). En revanche, seul le leader retourne une réponse (comme dans le cas de la réplication passive). Au cas où le leader défaille par arrêt, une réplique parmi l'ensemble des secondaires prend la relève. Cette approche est caractérisée par la rapidité du recouvrement et par la non nécessité de la mise à jour de l'état puisque toute les requêtes sont exécutées parallèlement. En revanche, le déterminisme de toutes les répliques est nécessaire puisque chacune d'entre elles gère la mise à jour de son état interne.

L'ensemble des caractéristiques, en termes d'hypothèses et de qualité de service, des différents modes de réplications est représenté dans les tableaux 1 et 2 :

Réplication / Hypothèses	Active	Semi-active	Passive
Déterminisme des applications	Oui	Oui	Non
Ordonnancement des messages	Oui	Non	Non
Accès à l'état	Non	Non	Oui
Fautes par valeurs	Oui	Non[3]	Non
Fautes temporelles	Oui	Oui	Oui

Tableau 1. Hypothèses pour la réalisation des différentes stratégies de réplication

Réplication / Qualité de service	Active	Semi-active	Passive
Temps de recouvrement	++	+	-
Chargement du réseau	-	+	++

Tableau 2. Influence du mode de réplication sur la qualité de service

Le Tableau 1 regroupe, d'une part, les caractéristiques requises par un système distribué pour pouvoir bénéficier d'un mode de réplication particulier (i.e. le déterminisme des applications, l'ordonnacement des messages et l'accès à l'état de l'application), et d'autre part, ce que les

[3] Une extension de la réplication sem-active est décrite dans [Powell 1991] permettant de pallier les fautes par valeurs.

différents modes de réplication offrent en termes de tolérance aux fautes (i.e. fautes par valeurs et fautes temporelles).

Le Tableau 2 montre l'influence du choix d'un mode de réplication sur la qualité de service d'un système distribué répliqué. En effet, la réplication active est la plus appropriée pour des systèmes à fortes contraintes temporelles (i.e. systèmes temps réel) puisqu'en cas d'occurrence d'erreur, elle offre les meilleurs délais de recouvrement. En revanche, elle est la plus exigeante en ressources, en particulier elle nécessite une large bande passante du réseau du fait de la résolution de certains problèmes, par exemple le consensus.

La réplication passive offre des performances inverses, c'est-à-dire un temps de recouvrement très lent et un chargement de réseau relativement bas. Lors du fonctionnement en absence de fautes, l'envoi des messages de synchronisation de l'état aux différentes répliques surcharge le réseau avec un faible coût. Mais lors de l'occurrence d'une défaillance, la mise à jour de l'état du nouveau primaire n'est pas immédiate.

Quant à la réplication semi-active, elle offre un bon compromis entre le temps de recouvrement (i.e. moins long que la réplication passive) et le chargement du réseau (i.e. moins chargé que la réplication active). D'une part, le temps de basculement est similaire à celui qui est offert par la réplication active, puisque les traitements redondants sont réalisés parallèlement par les différents membres du groupe de répliques, et d'autre part, comme les messages de synchronisation sont évités, la surcharge du réseau reste assez limitée.

Pour mettre en œuvre un système répliqué, nous devons avoir connaissance du nombre des répliques et assurer un ordre dans la transmission des messages. Ces fonctionnalités forment les éléments de base de la communication de groupe que nous allons présenter ci-après.

I.2.3.2. Systèmes de communication de groupe

La notion de groupe a été utilisée pour la première fois dans v-kernel [Cheriton 1985] pour gérer un ensemble de processus situés sur des machines différentes et qui coopèrent entre-elles. Cette notion a été étendue, par la suite, pour résoudre les problèmes liés à la réplication dans les travaux qui ont été réalisés autour du projet Isis [Birman 1993]. Si un système de communication de groupe (*Group Communication Service, GCS*) semble en première approche une commodité pour la mise en œuvre des systèmes répliqués, dans la mesure où il masque le nombre et la localisation des différentes répliques, il s'avère par la suite indispensable. En effet, il garantit plusieurs services incontournables pour cette mise en œuvre, à savoir la gestion du groupe des répliques, le consensus, l'ordonnancement des messages ou encore la mise à jour de l'état et de la reprise. Dans ce qui suit, nous présentons les différents services qu'il offre :

- **Gestion du groupe des répliques** : Outre la création d'un groupe, l'ajout d'un nouveau membre ou l'élimination d'une réplique défaillante, il peut offrir deux types de politiques :

- **Statique :** Le nombre de répliques est fixé au lancement du système, il ne change pas même en cas de défaillance d'une réplique. Le GCS est responsable de l'instantiation de nouvelles répliques se substituant à celles qui ont défailli.

- **Dynamique :** Le nombre de répliques peut changer au cours du fonctionnement du système pour augmenter le degré de redondance. Par ailleurs, cette deuxième politique est utilisée afin de permettre une évolution du service initial rendu. En effet, on double le nombre des répliques d'un groupe en ajoutant des répliques ayant de nouvelles fonctionnalités, puis on procède à l'élimination des anciennes répliques, pour ne pas perdre le degré de redondance initial. Ceci garantit une transparence de l'évolution du service rendu par le groupe par rapport aux anciens clients.

- **Consensus** : il permet à plusieurs répliques d'arriver à une décision commune malgré la possible défaillance de certaines d'entre elles. Le problème du consensus présente une abstraction pour résoudre des problèmes d'accord(s) relatif(s) à la tolérance aux fautes dans les systèmes distribués; par exemple, résoudre le problème de l'ordre total, de l'appartenance à un groupe de répliques ou tolérer une erreur par valeur retournée par une réplique. Pour qu'une décision soit le résultat d'un consensus:

  1)- chaque réplique correcte doit donner une décision, propriété de *Terminaison*.

  2)- toute les répliques doivent donner au plus une décision, propriété de *Intégrité uniforme*.

  3)- deux répliques correctes ne doivent pas donner deux décisions différentes, propriété de *Accord*.

  4)- Il faut que la décision *d* d'une réplique soit proposée aussi par d'autres répliques.

- **Ordonnancement des messages** : L'ordonnancement des messages est nécessaire pour la réplication active. En effet, pour que les différentes répliques aient des comportements identiques, le GCS assure un ordre total des messages sortants. Par ailleurs, dans le cas des systèmes à modèle temporel asynchrone, où il n'y a pas de bornes pour le temps de traitement et de transmission des messages, le GCS doit maintenir un synchronisme virtuel [Birman 1993]. Ainsi, les différentes répliques observent les changements de vues et reçoivent les différents messages dans le même ordre. Ces évènements sont synchrones en termes de temps logique et asynchrones en termes de temps physique.

- **Mise à jour de l'état et de la reprise** : A chaque fois qu'une nouvelle réplique rejoint le groupe, par exemple lors du clonage suite à une défaillance afin d'augmenter le nombre de répliques, elle doit mettre à jour son état interne pour avoir le même état

que les autres membres. La mise à jour de l'état d'une nouvelle réplique, peut s'effectuer selon les deux manières suivantes :

- L'état de la réplique est récupéré. Ceci implique que toute réplique fournit une première méthode de lecture de l'état et une seconde méthode de mise à jour de l'état (i.e. écriture).

- L'ensemble des requêtes effectuées par les répliques est enregistré, puis elles sont re-exécutées de manière déterministe sur la nouvelle réplique. Dans ce cas, la réplique n'est pas obligée de fournir une méthode de mise à jour de l'état, mais la procédure peut être bien plus longue, voire même impossible (impact sur l'environnement du système) selon les cas.

Nous avons présenté dans cette partie différentes notions de sûreté de fonctionnement. Nous avons commencé par définir la terminologie que nous allons utiliser dans ce mémoire pour traiter les aspects relatifs à la tolérance aux fautes. Ensuite, nous avons exposé les modes de défaillances et différents modèles temporels afin de présenter les hypothèses que nous allons utiliser dans nos travaux. Enfin, nous avons présenté les mécanismes de tolérance aux fautes (les mécanismes de réplications et les services fournis par un système de communication de groupe) que nous allons traiter dans les chapitres suivants. Dans la section qui suit, nous présentons la réflexivité qui permet de munir les systèmes informatiques de mécanismes non-fonctionnels, en particulier de tolérance aux fautes par réplication.

I.3. LA REFLEXIVITE

I.3.1. Introduction

« Est réflexif tout sujet capable d'appliquer à lui-même ses propres capacités d'actions » [Taiani 2004]. Les premiers travaux en informatique qui ont fait référence à la réflexivité remontent aux années 60. Ils ont été menés au niveau des langages de programmation comme le lambda-calcul et LISP. Vers la fin des années 70, les premières applications de cette notion se sont articulées autour de l'intelligence artificielle, domaine où l'on cherche à doter les systèmes informatiques d'une certaine autonomie. Cette autonomie se matérialise par une nouvelle forme de conception des systèmes, en définissant des axiomes (les actions élémentaires), des règles d'inférences à suivre et des heuristiques. En suivant cette nouvelle procédure, un système peut ajouter des règles, les ajuster ou même en éliminer certaines.

"A process's integral ability to present, operate on, otherwise deal with itself in the same way that it represents, operates and deals with its primary subject matter." [Smith 1984]

Cette définition évoque le pouvoir que doit posséder un système réflexif aussi bien pour appliquer des actions afin de résoudre les problèmes auxquels il est censé apporter une

solution, que pour modifier son comportement et sa structure, notion d'*auto-représentation*. Pour que cette modification puisse prendre effet, il faut que le modèle qui représente le système « simple » soit en relation causale avec le système lui-même. C'est-à-dire, tout changement du système provoque le changement du modèle, et tout changement du modèle introduit un changement du système et de son comportement.

"A reflective system is a computational system which is about itself in a casually connected way[4]*"* [Maes 1987b].

Par ailleurs, la réflexivité peut être vue comme une manière « élégante » d'organiser un système pour répondre d'une façon efficace aux problèmes qu'il doit résoudre.

Reflective computation does not directly contribue to solving problems in the external domain of the system. Instead, it contribues to the internal organization of the system or to its interface to the external world. Its purpose is to guarantee the effective and smolt functionning of the object computation [Maes 1987b].

L'utilisation d'une telle approche permet d'avoir une meilleure productivité. En effet, la réflexivité est un moyen puissant de réaliser un découplage entre les aspects applicatifs et les aspects non-fonctionnels *orthogonaux* d'un système. Ce pouvoir de découplage qui est appelé aussi *cross-cutting concerns* facilite :

- le développement des applications, puisqu'il permet de factoriser le code orthogonal,
- leur maintenance, puisqu'il permet au développeur de mettre moins de temps pour apporter les changements et diminuer les chances d'occurrence d'erreurs, et
- leur réutilisation, puisque le code de l'application est bien séparé.

En revanche, la réflexivité ne peut pas être considérée comme incontournable pour mettre en œuvre des systèmes informatiques, puisqu'on peut avoir un modèle d'organisation des applications sans découplage des aspects fonctionnels de ceux qui sont non-fonctionnels.

I.3.2. Terminologie

Comme on peut le voir sur la Figure 7, un système réflexif est composé de deux niveaux :

[4] A system is said to be causually connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other.

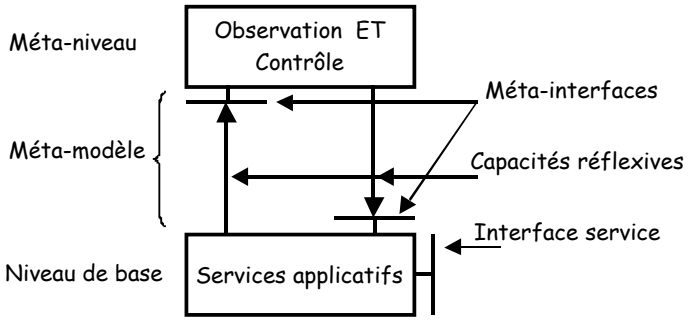


Figure 7. Architecture générale d'un système réflexif

Niveau de base : Il représente le lieu de *traitement de l'information* et est responsable de l'aspect fonctionnel du système. Il offre une interface dite *fonctionnelle* pour acquérir les requêtes des utilisateurs et leur fournir un service. Les utilisateurs peuvent être des systèmes ou d'autres composants du même système. Par ailleurs, le niveau de base offre une *méta-interface* à travers laquelle ce traitement peut être influencé par le méta-niveau.

Méta-niveau : Il représente le lieu de *l'interprétation* des informations relatives au niveau de base (arrivée d'une requête sur l'interface service, changement de structure, etc.). Ces informations sont perçues par le méta-niveau à travers sa *méta-interface*. Le méta-niveau représente, par ailleurs, le lieu de *l'influence* du comportement et de la structure du niveau de base lorsque le méta-niveau agit sur la *méta-interface* du niveau de base. Cette influence qui relève de l'aspect non fonctionnel du système se matérialise par la modification de la structure et du comportement du niveau de base. Le méta-modèle représente l'image à travers laquelle le méta-niveau interprète et modifie le niveau de base. Le méta-niveau peut observer et contrôler aussi bien la structure que le comportement du niveau de base. Cette capacité est rendue possible grâce aux trois mécanismes suivants :

Réification : C'est la capacité de notification au méta-niveau des évènements comportementaux et structurels du niveau de base. Cette capacité reflète le pouvoir d'observation du comportement du niveau de base par le méta-niveau.

Introspection : L'introspection reflète la capacité d'observation de la structure du méta-niveau. En effet, le méta-niveau utilise la méta-interface fournie par le niveau de base pour récupérer des informations relatives à son niveau fonctionnel (par exemple, la valeur d'un attribut).

Intercession : L'intercession relève du contrôle que peut avoir le méta-niveau sur le niveau de base. Comme pour le cas de l'observation, on peut distinguer :

1. *L'intercession comportementale*, par laquelle le méta-niveau peut modifier le comportement du niveau de base, par exemple en empêchant la délivrance d'une réponse à des fins de sécurité.

2. *L'intercession structurelle*, par laquelle le méta-niveau peut intervenir sur la structure du niveau de base, par exemple en ajoutant de nouvelles méthodes dans l'interface fonctionnelle.

I.3.3. Le déploiement de la réflexivité

Comme le montre la Figure 8, un système informatique est composé d'une application, d'une couche intermédiaire, d'un système d'exploitation et d'une couche matérielle.

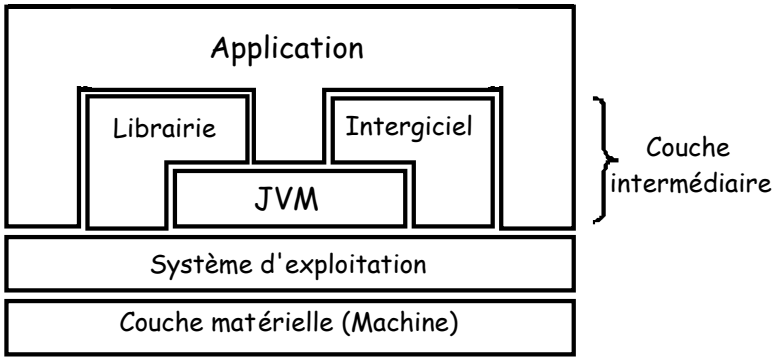


Figure 8. Différentes couches d'un système informatique

L'application est écrite dans un langage spécifique, et compréhensible par le développeur du service qu'elle doit fournir. Par ailleurs, cette application peut être composée de plusieurs applications qui dans leurs interactions et collaborations fournissent un service complexe.

Dans le but de dispenser le développeur de l'application des spécificités du support sous-jacent, la couche intermédiaire, qui est composée de librairies et d'intergiciels, fournit des interfaces qui cachent les primitives des systèmes opératoires et de communications.

Le système opératoire représente le support d'exécution de toute application en translatant les actions décrites dans un langage évolué en primitives compréhensibles au niveau de la machine. Cette translation est réalisée par le biais d'une interface qui est différente de celle fournie par le matériel.

La couche matérielle, représente le support physique du traitement des données gérées par le système d'exploitation (i. e. stockage et calcul des données).

Nous présentons dans la suite l'application de la réflexivité aux différents niveaux des systèmes informatiques.

I.3.3.1. Réflexivité au niveau applicatif

À ce niveau, la réflexivité permet de munir une application « standard » de spécificités relatives à la tolérance aux fautes comme la réplication, à la sécurité comme le chiffrement des messages, ou encore à la maintenance corrective comme la mise à jour de nouvelles fonctionnalités sans dégrader la disponibilité du système.

L'objectif du méta-niveau est de pouvoir observer et modifier le comportement et la structure d'une application (niveau de base) à l'exécution. On parle d'exécution « souple » si les applications sont écrites dans un langage interprété (cf. Figure 9.a) ; celles-ci nécessitent un programme auxiliaire appelé « interpréteur » qui exécute les instructions du langage dans lequel elles sont écrites ou d'un langage intermédiaire résultat de la traduction du langage de l'application. Cette « souplesse » réside dans la possibilité de changer le comportement d'une exécution, en modifiant l'interpréteur et sans toucher au code de l'application. En revanche, on parle d'exécution « rigide » si les applications sont écrites dans un langage compilé (cf. Figure 9.b) ; celles-ci nécessitent un programme auxiliaire appelé « compilateur » pour générer un programme final, écrit en langage « machine » et appelé « exécutable ». Ce programme final est compréhensible par le processeur, ce qui le rend indépendant de tout interpréteur. La rigidité réside dans la génération obligatoire d'un exécutable qui ne peut plus être modifié. Ainsi, si nous voulons changer le résultat d'une exécution nous devons modifier le code de l'application ou le compilateur, et toute modification, du code ou du compilateur, est suivie d'une génération d'un nouvel exécutable.

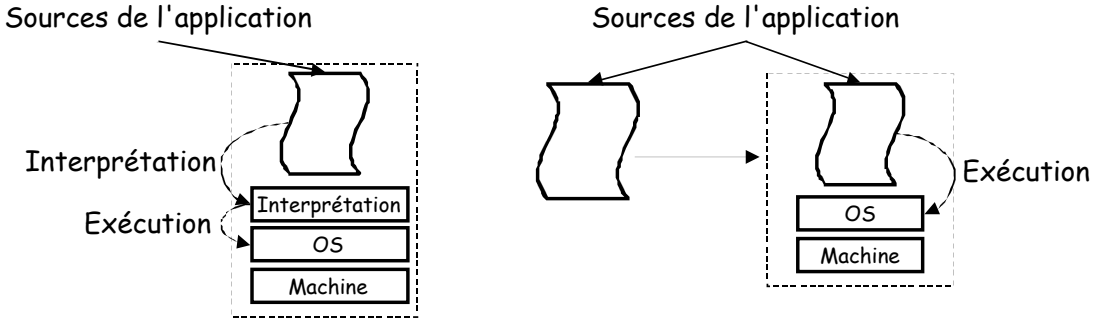


Figure 9.a. Langage interprété **b. Langage compilé**

Réflexivité dans les langages interprétés

Les premiers travaux menés dans le but de rendre réflexive une application procédurale ont été développés avec des langages interprétés. Dans ce type de langage, l'application joue le rôle du niveau de base et l'interpréteur joue le rôle du méta niveau. Par ailleurs, comme un interpréteur accède au code de l'application lors de l'exécution pour traduire le code de l'application, il peut lire ou modifier les valeurs des variables locales (respectivement globales), d'une procédure (respectivement d'une application), ce qui l'affranchit du besoin d'un méta-modèle pour la structure de l'application.

Selon les travaux de Smith sur les langages procéduraux [Smith 1982], une architecture réflexive peut être vue comme une *tour* infinie d'étages réflexifs (cf. Figure 10) où chaque niveau interprète celui qui le précède ; en effet, cette tour est définie au-dessus d'une application, écrite dans un langage interprété. Le premier étage représente l'interpréteur méta-circulaire (Meta-Circular Processor, MCP) I_1, écrit dans le même langage que celui de l'application, il interprète l'application, et est interprété par l'interpréteur du dessus (I_2). Ce dernier est lui aussi un MCP, dans la mesure où il est interprété par l'interpréteur (I_3), etc…

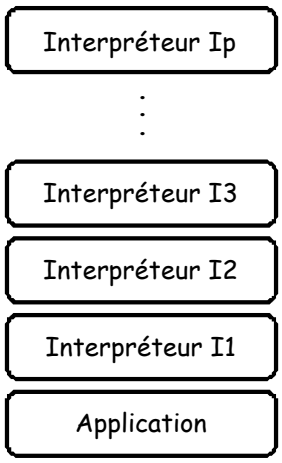


Figure 10. Tour infinie d'interpréteur méta-circulaires

En pratique, pour pouvoir être exécutée par le processeur, cette infinité de niveaux nécessite un interpréteur primitif appelé (Ip). Par ailleurs, la notion d'empilement d'interpréteurs est très intéressante dans la mesure où cette tour réflexive peut permettre de spécialiser l'interprétation pour un besoin non fonctionnel. Ainsi, nous pouvons définir un premier MCP relatif au de-verminage de l'application (I_{deb}), ensuite, nous rajoutons un second niveau relatif à la tolérance aux fautes (I_{taf}), et enfin, un troisième interpréteur méta-circulaire de sécurité (I_{sec}) (cf. Figure 11).

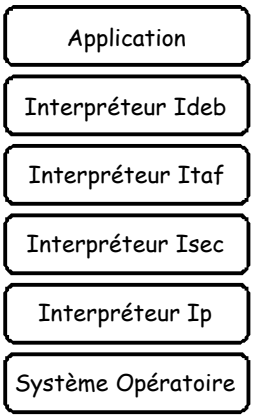


Figure 11. Exemple d'une tour finie d'interpréteurs méta-circulaires

Réflexivité dans les langages compilés

Malgré le fait que l'exécution des applications compilées soit rigide ; on ne peut pas modifier l'exécution d'une application après avoir compilé son code ; la théorie de Smith reste encore valable. En effet, considérons « A » le code d'une application écrite dans un langage compilé, et C1 le processus de compilation, c'est-à-dire la transformation de *A* en un code exécutable E_1. Ce dernier code peut être à son tour compilé par un second compilateur C2, qui génèrera un nouveau code exécutable E_2 différent de E_1. Ainsi, nous pouvons imaginer une succession de compilateurs C3, C4, …, Cn pour obtenir, à la fin, un code exécutable E.

Cette théorie n'a pas été développée dans le cadre des langages compilés et orientés objet. En revanche, elle a été partiellement adoptée dans le cadre du logiciel OpenC++V1 [Chiba 1993]; Elle a été utilisée d'une manière *simple* (i.e. seuls deux niveaux de compilation ont été introduits) et *statique* (i.e. une seule façon de transformer le code est possible). Par ailleurs, il est à noter que dans la deuxième version de ce logiciel [Chiba 1995], la transformation du code est rendue dynamique, en la spécifiant à travers une méta-classe[5], ce qui montre un rapprochement avec la théorie de Smith.

Réflexivité dans les langages à objets

La réflexivité a été introduite dans les langages orientés objets *LOO* afin d'enrichir leur pouvoir d'expression. En effet, les langages orientés objets sont particulièrement performants pour exprimer les fonctionnalités des applications. La réflexivité appliquée à ceux-ci, permet de se concentrer sur des aspects non-fonctionnels ayant trait aux entités de « base » des LOO : classes, objets, attributs, méthodes. Elle permet donc de raisonner à un méta-niveau concernant ces entités ; par exemple : la création d'un objet, l'invocation d'une méthode etc. La réflexivité a influencé le développement et l'évolution de plusieurs langages orientés objets, par exemple ObjVLisp [Cointe 1987] et SMALLTALK-80 [Foote 1989].

La coexistence du code relatif à l'aspect fonctionnel et non fonctionnel au niveau de l'objet va à l'encontre de la modularité que les langages orientés objets tendent à apporter. Ainsi, plusieurs travaux ont essayé de séparer ces aspects dans des niveaux différents qu'ils ont appelé niveau objet et niveau réflexif dans le langage 3-KRS [Maes 1987a], puis niveau de base et méta-niveau dans le langage CLOS [Kiczales 1991]. L'ensemble du système étant orienté objet, le méta-niveau d'un système à la fois orienté objet et réflexif est lui aussi structuré sous forme d'objets particuliers, appelés méta-objets, qui encapsulent les aspects réflexifs du système. Ces méta-objets interagissent avec le niveau de base, organisé en objets de base, en utilisant les mécanismes de réification, d'introspection, et d'intercession que nous avons mentionnés. Du fait de la structuration en termes d'objets, on parle alors de protocole à méta-objets (*Meta-Object Protocol, MOP*) pour désigner l'ensemble des conventions qui régissent les interactions entre les méta-objets et objets de base, notamment en termes d'association (comment les méta-objets sont-ils créés ? détruits ? comment leur cycle de vie interagit-il avec celui des objets de base ?).

I.3.3.2. Réflexivité au niveau intermédiaire

La réflexivité au niveau intermédiaire a pour rôle de rendre cette couche flexible et ajustable. Cette flexibilité permet d'adapter le niveau intermédiaire afin de répondre aux évolutions des applications et des systèmes sous-jacents (i.e. système opératoire et couche physique).

---

[5] Une méta-classe est un programme capable de personnaliser le comportement d'un compilateur. Ainsi, on peut changer le comportement d'un compilateur en changeant de méta-classe.

Le niveau intermédiaire dans les architectures de systèmes informatiques multi-couches se compose de logiciels de nature différents, tels que :

- des librairies qui fournissent des briques de base sous la forme de méthodes ou d'objets facilitant le développement des applications et que l'utilisateur doit lier à sa propre implémentation,
- des machines virtuelles [Lindholm 1999], qui assurent l'indépendance de l'application du système opératoire et de la machine (i.e. couche physique). Cette indépendance rend les applications portables,
- et/ou des intergiciels, qui facilitent le développement des applications distribuées, en masquant par exemple l'aspect gestion de la communication (en termes de création de la connexion, envoi des requêtes, etc.).

Les librairies

Les librairies réflexives que nous pouvons trouver dans les plates-formes de développement, par exemple .NET de Microsoft ou J2EE de Sun microsystems, fournissent aux concepteurs la possibilité de doter leurs applications de mécanismes d'observation et de contrôle comportemental et structurel standard.

En effet, dans *.NET*, la librairie réflexive, qui se trouve dans l'espace de noms `system.Reflection`, est composée de classes et d'interfaces permettant l'obtention d'une vue sur les types, les méthodes et les champs chargés. Elle permet, en outre, la création et l'appel dynamique de types. En se basant sur ces notions de la réflexivité, en particulier sur l'aspect introspection, le projet *fxCop* [Microsoft 2004] offre un outil d'analyse du code des applications pour détecter les carences en termes de conformité par rapport aux règles de conceptions et d'implémentations spécifiées par *.NET*. Ainsi, on implémente ces règles de conformités dans *fxCop* (méta-niveau) qui seront appliquées à l'application (niveau de base) pour détecter d'éventuelles erreurs.

Par ailleurs, dans la J2EE les librairies réflexives s'étendent sur plusieurs espaces de noms. Par exemple, `java.util` fournit le mécanisme de réification à travers la classe *observable* (niveau de base) qui notifie tout changement relatif à son état à un ou plusieurs observateurs (méta-niveau) qui doivent implémenter l'interface *observer*.

Malgré la séparation des codes fonctionnels (application) et non-fontionnels (méta-niveau), la réflexivité introduite au niveau des librairies souffre de la non-transparence de ces mécanismes. Ceci est dû à leur mode d'activation qui reste explicite et à la charge du développeur de l'application ; ce qui rend difficile la mise au point et la maintenance de tels systèmes.

Les machines virtuelles

Plusieurs travaux concernant la réflexivité ont eu pour cible des systèmes qui font appel à des machines virtuelles, en particulier la machine virtuelle java (JVM). Ces systèmes ont la particularité d'avoir un code qui subit plusieurs transformations pendant son cycle de vie avant d'être exécuté. En effet, le code d'une application est, dans un premier temps, compilé sous forme d'un fichier (par exemple *.class* pour JAVA) renfermant le *byte code*, i.e. un pseudo code. Ce nouveau code est, par la suite, chargé au niveau de la machine virtuelle par le biais du chargeur de classes (class loader) pour être exécuté, ou compilé, une seconde fois, en un code machine spécifique à la machine hôte.

Une synthèse proposée par Ian Welch [Welch 1999] montre les différents travaux qui ont traité cette problématique, en précisant à quelle niveau l'aspect réflexif est introduit dans le cycle de vie de ces systèmes, leurs possibilités et leurs limites en termes de pouvoir réflexif. Les machines virtuelles réflexives sont celles qui appliquent la réflexivité d'une manière dynamique lors de leurs exécutions. En effet, les travaux sur MetaXa de l'université d'Erlangen [Golm 1998], où un méta-objet peut être attaché non seulement à un objet de l'application, mais aussi, à des composants de la machine virtuelle elle même, peut agir sur le comportement de la JVM. Par exemple, on peut envisager d'attacher un méta-objet :

- au chargeur de classes (*classloader*) pour mettre en œuvre différentes stratégies de chargement,
- au moniteur pour mettre en œuvre différentes politiques de verrouillage (basées sur la priorité ou non),
- au niveau des brins d'exécutions, pour recevoir un évènement si un code opération (opcode) spécifique est exécuté ; le méta-objet peut implémenter l'opcode d'une manière différente de la machine virtuelle.

Les intergiciels

Les intergiciels réflexifs exploitent la position incontournable qu'ils occupent dans l'acheminement des requêtes entre les différents composants d'une application distribuée. Ils peuvent observer et contrôler le comportement des applications d'une part et adapter le traitement des requêtes (comportement) d'autre part. Par exemple, dans Reflective ORB [Ashish 1997], on définit une architecture minimale d'ORB sur laquelle on greffe un invocateur du côté client, et un expéditeur du côté serveur. L'invocateur a pour rôle d'ajouter de la méta-information au niveau de la requête avant qu'elle soit sérialisée (*marshalling*) ; ces méta-données peuvent par exemple définir un délai pour le traitement des invocations temps-réel. Une fois que la requête arrive du côté serveur, l'expéditeur peut par exemple vérifier si le traitement de la requête respecte les contraintes introduites par l'invocateur, ce qui pourra influencer le comportement du serveur. Ces travaux ont mené à la définition d'un *quarterware* (une sorte de librairie) pour les intergiciels, qui fournit plusieurs types d'invocateurs et d'expéditeurs (i.e. temps réel, tolérance aux fautes, gestion de groupe, etc.) pouvant influencer

le comportement de l'application ainsi que sa propre architecture et le mode de traitement des requêtes [Ashish 1998].

D'autres aspects qui traitent d'une part de l'observation et de la modification dynamique de l'architecture interne des intergiciels réflexifs seront présentés dans le chapitre II. Ces aspects sont d'une importance capitale pour des systèmes qui doivent s'auto-configurer lorsqu'ils détectent un changement au niveau de leur infrastructure d'exécution (i.e. basculement d'un mode de communication câblée à un mode de communication sans fil).

I.3.3.3. Réflexivité aux noyau et système opératoire

Les composants de bases d'un système opératoire (noyau, système de gestion de la mémoire et système de fichiers) ont pour rôle la protection et la gestion de la couche physique. Par exemple, la gestion de la mémoire virtuelle afin d'allouer la mémoire physique, rôle confié au gestionnaire de mémoire, relève de la protection des ressources. Tandis que, l'allocation du ou des processeurs, le traitement des interruptions et la gestion des horloges, qui est à la charge du noyau, relève de la gestion des ressources physiques. Néanmoins, l'utilisation effective d'un système d'exploitation nécessite d'autres sous-systèmes qui n'en font pas partie, par exemple un interprète de langage de commande, un gérant des fenêtres et de réseau et une bibliothèque d'exécution pour les langages de programmation.

Historiquement, les premiers systèmes d'exploitations étaient monolithiques : les composants de base ne sont pas des modules séparables, mais sont étroitement intégrés dans une structure unique (noyau). Cette organisation permet d'atteindre de hautes performances en accélérant les communications internes, mais elle rend difficile l'évolution et l'adaptation des systèmes. Ces considérations ont conduit à définir une nouvelle architecture de systèmes d'exploitation, dans laquelle un *micro-noyau* concentre les fonctions élémentaires de gestion du processeur, de la mémoire et des communications, les autres composants (gérant de processus, de mémoire virtuelle, de fichiers, de réseau, etc.) étant réalisés par de serveurs inter-communicants utilisant les fonctions du micro-noyau.

Malgré ces nouveaux acquis, ces systèmes présentent de sérieuses limitations, puisque, seuls les serveurs privilégiés et le noyau ont le droit de gérer les ressources du système. Les autres applications, en évolution continue, sont limitées aux interfaces fournies par ceux-ci et qui sont figées. Ainsi, pour pallier ces limitations, l'architecture des systèmes opératoires se voit dans l'obligation de fournir une interface qui anticipe les besoins de nouvelles applications, ce qui n'est pas envisageable sur le plan pratique. Deux grands axes ont été exploités pour fournir des systèmes opératoires réflexifs : le développement vertical et le développement horizontal.

Développement vertical

Cette approche s'appelle aussi la machine virtuelle[6] . Elle a été proposée initialement par IBM [Creasy 1981] et définit une machine virtuelle minimale (VM OS), comme la VM/370, qui émule et contrôle l'accès aux ressources physiques, par exemple l'architecture du S/370. Chaque application peut par la suite définir sa propre machine virtuelle qui émule la machine virtuelle minimale. Dans le projet VM/370, on distingue deux niveaux de machines virtuelles. Dans les travaux de Golberg [Golberg 1983], on définit d'une manière formelle la possibilité de concevoir une tour de machines virtuelles, où chaque niveau émule celui qui le précède. Le terme « développement vertical » vient du fait que les besoins d'une nouvelle application sont décrits au niveau d'une nouvelle machine virtuelle se situant au-dessus de celle initiale qu'elle émule. Par ailleurs, cette technique a été reprise dans le noyau (*User Mode Linux*, UML) afin de permettre l'exécution de programmes renfermant des erreurs, de les expérimenter avec de nouvelles versions du noyau Linux et même de programmer des commandes au niveau interne du noyau sans risque d'endommager l'installation du système opératoire.

Développement horizontal

Deux approches sont à distinguer pour le développement horizontal des systèmes d'exploitations réflexifs : l'approche micro-noyau et l'approche exo-noyau.

L'approche appelée micro-noyau (exemple : projet SPIN [Bershad 1995]) est basée sur le chargement de code au niveau noyau. Elle est conceptuellement proche de l'approche horizontale à l'exception du fait que le code que nécessite une application ne se trouve plus au niveau de la machine virtuelle, mais au niveau du noyau. En effet, ce système opératoire est composé de services centraux et d'autres d'extensions. Ces derniers permettent le chargement en tout moment des extensions, au niveau du noyau, qui s'autointègrent pour fournir les services spécifiques que demande l'application qui les a déployés. Cette approche, utilise l'encapsulation et la sécurité offertes par le langage Modula-3 [Nelson 1991] pour protéger le noyau des extensions non valides et implémenter une communication protégée en utilisant l'appel de procédures.

L'approche appelée exo-noyau (exemple : projet Aegis [Engler 1995]) est basée sur la définition d'un système opératoire minimal dont la fonction est de protéger l'utilisation de la couche physique. La partie dédiée à la gestion des ressources, est confiée à l'application qui les utilise. Cette approche semble ressembler à l'approche micro-noyau dans la mesure où elle fixe une première couche privilégiée et une deuxième non privilégiée ; par contre, elle se distingue par le fait qu'elle ne confie à la couche privilégiée que le rôle de protection, toutes

---

[6] Le simulateur de machine virtuelle (virtual machine simulator), par exemple l'interpréteur JAVA, est un programme qui s'exécute sur une machine d'une architecture donnée en implémentant une machine virtuelle avec une architecture totalement différente de celle de la machine. En revanche, le moniteur de machine virtuelle (virtual machine monitor ou hypervisor), par exemple la machine virtuelle VM/370 de IBM, est un programme qui crée un ou plusieurs machines virtuelles qui exportent l'architecture de la machine sur laquelle ils s'exécutent.

les autres tâches (définition de la gestion de l'allocation des ressources, traitement des erreurs, pagination etc…) sont déléguées à la couche non privilégiée.

Si nous nous focalisons sur la couche du système d'exploitation, nous remarquons que dans ces différents travaux, le système proposé est formé de deux parties : une couche de base qui renferme, les fonctionnalités irréductibles que doit fournir un système opératoire, et une deuxième couche, qui renferme les fonctionnalités que doit fournir un OS pour qu'une application puisse s'exécuter.

Nous avons présenté dans cette section la notion de la réflexivité dans les systèmes informatiques ainsi que sa terminologie et les mécanismes qui lui sont associés. Nous avons montré l'étendue de l'utilisation de la réflexivité à différents niveaux (applicatif, intermédiaire et système opératoire) afin de situer nos travaux qui, quant à eux, se situent au niveau intermédiaire.

I.4. PROBLEMATIQUE ET APPROCHE RETENUE

Plusieurs travaux ont eu pour objectif de munir une application distribuée de mécanismes de tolérance aux fautes d'une manière réflexive. MAUD [Agha 1993] est le premier projet qui s'inscrit dans ce cadre. Son architecture utilise le modèle à acteurs, où les différents composants coopèrent en échangeant des messages. MAUD, qui utilise le langage à acteurs HAL, 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 méta-objets).

D'autres projets ont ensuite suivi. A titre d'exemple, GARF [Guerraoui 1997] est basé sur l'aspect réflexif proposé par le langage SmallTalk. Il exploite le fait que toute requête doit, avant d'être émise, passer par un mandataire *proxy*. Ce proxy a la particularité de n'implémenter aucune méthode. Ainsi lorsqu'une requête est invoquée, elle est tout d'abord vérifiée dans le dictionnaire de méthodes du proxy. En réaction, ce dernier soulève l'exception `DoesNotUnderstand` à une classe du système (méta-niveau), appelée ClassName, qui peut la rediriger, la bloquer ou tout simplement tracer son émission.

Nous remarquons que ces deux travaux, MAUD et GARF, exploitent le pouvoir réflexif offert par les langages HAL et SmallTalk pour mettre en œuvre la tolérance aux fautes. Ainsi nous identifions la réflexivité au niveau de la couche applicative. Ces approches qui sont basées sur un langage particulier ne sont pas standard et donc très peu portables.

Contrairement à ces deux travaux, de nouvelles études [Karablieh 2002] et [Taiani 2004] ont montré l'intérêt d'utiliser la réflexivité à différents niveaux d'un système informatique. Ils profitent, ainsi, du pouvoir d'expression des différents niveaux, à savoir schématiquement une forte sémantique pour les niveaux les plus hauts et une grande quantité d'informations qu'on peut récupérer des niveaux les plus bas. Même si ces nouvelles approches sont utilisables, leur

introduction sur le plan industriel, se heurte aux problèmes de normalisation et de standardisation.

Nous avons présenté dans la première partie de ce chapitre l'intergiciel CORBA sous différents angles : (1) la vue générale, où nous avons résumé les différents concepts qu'introduit cet intergiciel, comme l'objet distant, sa référence, son interface, etc. (2) la vue architecturale permet d'isoler les différents composants du bus à objet CORBA et qui sont utilisés dans (3) la vue du développeur. Nous avons présenté dans ce troisième point de vue, la façon avec laquelle est développée une application distribuée avec cet intergiciel. Avant de montrer les différentes transformations que subit une requête lors de son émission par le client jusqu'à son arrivée au niveau du serveur, nous avons exposé (4) les différents protocoles utilisés par cet ORB. Cette première partie a été conclue par la localisation des zones vulnérables et assujetties à des défaillances.

Nous avons présenté dans la seconde partie de ce chapitre les moyens nécessaires pour remédier aux défaillances qui peuvent survenir dans un système distribué. Après une énumération des différentes notions de base de la tolérance aux fautes, nous avons exposé les différentes hypothèses à prendre en compte pour pouvoir mettre en œuvre une solution appropriée fournissant un système qui livre son service même lors de l'occurrence de défaillances. Ces hypothèses sont relatives d'une part au modèle de communication auquel obéit le système, et d'autre part aux différents types de fautes que nous considérons. Cette seconde partie se termine par la présentation des différentes techniques de réplication de communication de groupe.

Dans la troisième partie de ce chapitre, nous avons présenté la notion de réflexivité qui peut être utilisée comme moyen élégant de mise en œuvre de la tolérance aux fautes dans les systèmes répartis. Nous avons présenté sa terminologie puis son application dans le cadre des systèmes informatiques. Nous avons, par la suite, dénombré trois niveaux, à savoir le niveau applicatif, intermédiaire et noyau, dans lesquels cette notion peut être déployée.

Nos travaux, qui s'inscrivent dans le même cadre que ceux de [Karablieh 2002] et [Taiani 2004], exploiteront la réflexivité inter-niveau, en particulier au niveau intermédiaire, pour munir les applications distribuées de mécanismes de tolérance aux fautes. Cependant, nous allons adopter une démarche différente de celle qui est utilisée précédemment. En effet, pour ne pas avoir à définir de nouveaux concepts et à les faire accepter par la communauté industrielle, nous exploiterons le potentiel réflexif des intergiciels standards, bien implantés dans le secteur industriel, pour montrer le besoin de les améliorer ou de définir de nouveaux concepts. L'intérêt d'une telle démarche réside dans la minimisation du coût d'acquisition et de mise en œuvre d'une nouvelle technologie.

Étant largement utilisé pour le développement d'applications distribuées industrielles, l'intergiciel CORBA fera l'objet de notre étude. Nous allons montrer le besoin d'améliorer ce support pour permettre une meilleure mise en œuvre des mécanismes de tolérance aux fautes. Nous allons identifier, dans un premier temps, le potentiel réflexif de l'intergiciel CORBA, aussi bien au niveau comportemental que structurel. Ensuite, nous évaluerons ce potentiel en

termes d'observation et de contrôle du comportement et de la structure des applications. Par la suite, nous définirons une plate-forme fournissant des mécanismes de réplication aux applications distribuées afin de montrer la difficulté d'une telle mise en œuvre et de justifier le besoin d'améliorer l'intergiciel actuel. Enfin, nous proposerons une amélioration du potentiel réflexif de l'intergiciel standard CORBA qui, d'une part, permettra une meilleure mise en œuvre de la tolérance aux fautes et, d'autre part, restera en concordance avec les anciennes applications.

Contrairement aux travaux antérieurs, où les développeurs des applications interviennent dans la mise en œuvre des mécanismes de tolérance aux fautes, notre objectif sera aussi de faire évoluer les intergiciels afin de fournir des outils permettant une mise en œuvre transparente des mécanismes non-fonctionnels. Le développement d'une plate-forme obéissant à cette contrainte, en utilisant les mécanismes actuels, nous renseignera sur leurs limites et nous permettra d'en déduire les améliorations à apporter à la norme. Ainsi, notre plate-forme sera composée de COTS qui seront transparents, aussi bien dans la mise en œuvre des mécanismes non fonctionnels que dans leur activation. Par ailleurs, cette plate-forme sera indépendante des applications qui les utiliseront puisqu'elle se basera sur des mécanismes génériques.

CHAPITRE II

SYSTEMES TOLERANT LES FAUTES A BASE D'INTERGICIELS

Si l'avènement des intergiciels de communication a profité au développement des applications réparties, il a aussi soulevé la question de leur utilisation dans le cadre d'applications réparties tolérant les fautes. En utilisant les notions de bases présentées dans le chapitre précédent, à savoir les différents composants d'un intergiciel, les mécanismes de réplications et la notion de réflexivité, nous présentons dans ce chapitre comment certains travaux avaient procédé pour répondre à la question de tolérance aux fautes. On peut classer les systèmes tolérant les fautes à base d'intergiciel en deux grandes familles : les systèmes classiques et les systèmes réflexifs.

Dans un premier temps, nous présentons les approches s'inscrivant dans le cadre de la première famille en les classant en fonction de leurs caractéristiques macroscopiques et intrinsèques. Les approches réflexives feront l'objet de la deuxième partie, où nous évaluerons plus particulièrement leur adaptabilité à l'implémentation des mécanismes de tolérance aux fautes. Ces différentes approches ont été à l'origine de la définition de la norme FT-CORBA que nous présentons dans la dernière partie. Cette partie présentera aussi notre point de vue sur cette nouvelle norme par rapport à la manière avec laquelle elle considère la tolérance aux fautes.

II.1. LES APPROCHES CLASSIQUES

Avant l'adoption de la norme FT-CORBA (Fault Tolerant CORBA) en 1999 [OMG 2002b], plusieurs projets ont eu pour objet de munir les intergiciels, implémentant la norme CORBA standard, de mécanismes de tolérance aux fautes, à savoir l'utilisation d'un système de communication de groupe, d'algorithmes de consensus ou de politiques de réplication. L'objectif était de définir une plate-forme qui serait utilisée d'une façon standard pour le développement d'applications distribuées tolérant les fautes. Ces premiers travaux étaient confrontés à deux dilemmes. Le premier est celui de la *spécificité* par rapport à la *portabilité* ; en effet, certains projets ont essayé d'exploiter la spécificité d'un système d'exploitation particulier ce qui nuit à la portabilité de la solution proposée. Le second est celui du choix de

la *transparence* par rapport à l'*interopérabilité*. En effet, certains travaux avaient pour objectif de fournir les mécanismes de tolérance aux fautes d'une manière transparente pour les applications ; cette transparence nécessite une certaine intrusion (c'est-à-dire, une modification de l'implémentation) au niveau de l'intergiciel. Ceci limite l'interopérabilité requise par la norme CORBA entre les ORB standard et ceux qui sont tolérants aux fautes.

II.1.1. Définition d'une classification

La notion d'un Système de Communication de Groupe (GCS) constitue un service de base pour mettre en œuvre des applications distribuées tolérantes aux fautes, par exemple par réplication active (voir la section 2 du chapitre I). En 1997, une première classification proposée par Felber [Felber 1998a] était dictée par l'emplacement du GCS ou la façon avec laquelle il était utilisé. Cette classification propose trois catégories d'approches différentes :

- Les approches par *intégration* ont été les premières à être utilisées. Dans ces approches, le code de l'ORB est modifié pour pouvoir supporter les mécanismes offerts par un GCS ou même intégrer un GCS du commerce à l'instar d'ISIS ou Totem (cf. Figure 12.a). Le principal problème soulevé par ces approches est l'intrusivité par rapport à l'ORB, sous la forme de modification au niveau de leur protocole de communication, ce qui limite leur interopérabilité. En effet, un serveur utilisant de tels ORB ne pourra pas fournir de service à un client utilisant la version standard du même ORB.

- Les approches par *services* sont proposées pour pallier le problème de l'interopérabilité. En effet, dans ces approches, le GCS est présenté comme un service implémenté au dessus de l'ORB. Ainsi l'ORB n'est pas modifié et peut être utilisé par des serveurs différents (cf. Figure 12.b). Si ces approches ont résolu le problème de l'interopérabilité, elles n'apportent pas de solution au problème de la transparence à cause de la délégation de l'utilisation des mécanismes de tolérance aux fautes aux serveurs.

- Les approches par *interception* sont proposées pour résoudre les problèmes de la transparence et d'interopérabilité. En effet, le GCS est présenté comme un service qui peut être inclus ou non par l'administrateur du système, d'une manière indépendante des serveurs. L'utilisation de ce service est assurée par la re-direction des requêtes sortantes du client vers le GCS. Ce dernier sera responsable de la mise en œuvre de la technique de réplication demandée par l'administrateur (cf. Figure 12.c). Comme l'interception des requêtes est réalisée au niveau d'un système d'exploitation spécifique, ces approches se trouvent confrontées au problème de la portabilité.

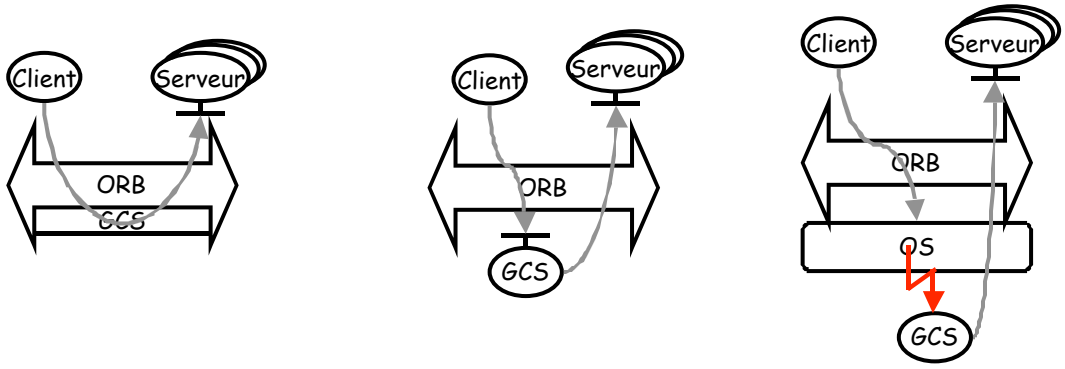


Figure 12. a. Intégration b. Service c. Interception

Pour pouvoir émettre un message à un groupe d'objets (répliques), un client a besoin d'un service qui lui permette de *diffuser* son message, le GCS *Group Communication Service*. Dans la classification que nous venons de présenter, l'aspect « microscopique » relatif à la façon avec laquelle un message est *diffusé* ne peut pas être distingué. Ainsi, nous avons relevé la nécessité d'ajouter une nouvelle dimension qui représente la manière avec laquelle la diffusion d'un message est demandée. Si cette demande est *explicite* par le client cela impliquera ce dernier dans le protocole de tolérance aux fautes. En revanche, dans le cas d'une demande *implicite*, le client ne fera pas partie du protocole mis en jeu. Nous présentons ci-après ces deux propriétés pour illustrer le degré d'implication des clients dans la mise en œuvre de la tolérance aux fautes, et donc le niveau de transparence/intrusivité de chaque solution.

La Figure 13 montre le fonctionnement des approches implicites.

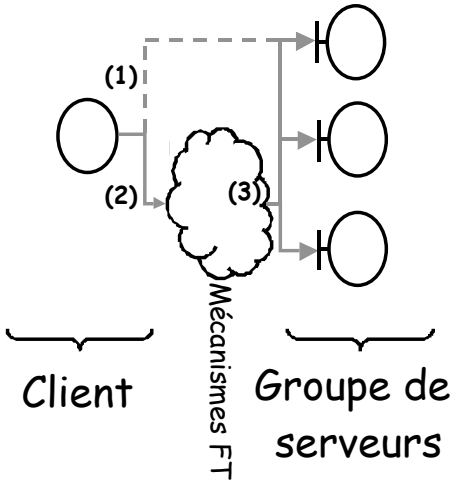


Figure 13. Approches Implicites

Dans cette approche, le client s'adresse d'une manière transparente aux mécanismes de tolérance aux fautes (2) comme s'il s'adressait à l'interface du serveur cible (1). En pratique, le client ne fait aucune différence entre un serveur répliqué et un qui ne l'est pas, puisqu'un objet CORBA est désigné par un IOR (Inter-orb Object Reference) qui ne peut désigner qu'un et un seul objet [OMG 2002a]. Dans ces approches, l'IOR a été modifié pour référencer un groupe

d'objets plutôt qu'un objet unique. Ces approches utilisent donc l'information contenue dans un IOR pour *diffuser* la requête aux différents membres du groupe (3).

La Figure 14 montre le fonctionnement des approches explicites.

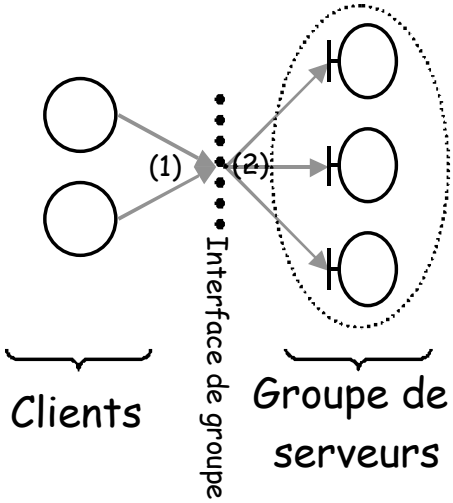


Figure 14. Approches Explicites

Dans cette approche, les clients s'adressent explicitement à l'interface du service de groupe (1) et sont dans une certaine mesure responsables d'une partie du protocole de tolérance aux fautes. En effet, les IOR des différents serveurs ne sont pas utilisées par les clients. Ces derniers obtiennent explicitement l'IOR relatif à un objet qui représente le GCS. Ce représentant peut être un mandataire qui redirige les requêtes au GCS, ou l'interface du service de groupe lui-même qui va diffuser les requêtes aux différents membres du groupe (2).

Nous allons présenter dans ce qui suit quatre projets, parmi ceux qui ont traité la tolérance aux fautes dans les systèmes répartis à base d'intergiciel CORBA. Ces travaux sont représentatifs de l'état de l'art puisqu'ils appartiennent aux trois classes d'approches définies par Felber. De plus, deux parmi-eux permettent de montrer la limite de cette classification et le besoin de l'affiner.

II.1.2. Approches implicites

II.1.2.1. Orbix+Isis

Orbix+Isis [IONA 1994] est une des premières solutions proposées pour fournir de la tolérance aux fautes à des applications distribuées basées sur un intergiciel CORBA. Cette approche est intrusive par rapport à l'intergiciel, dans la mesure où ce dernier est muni des fonctionnalités relatives à la tolérance aux fautes ; c'est cette caractéristique qui explique son classement parmi les approches par intégration.

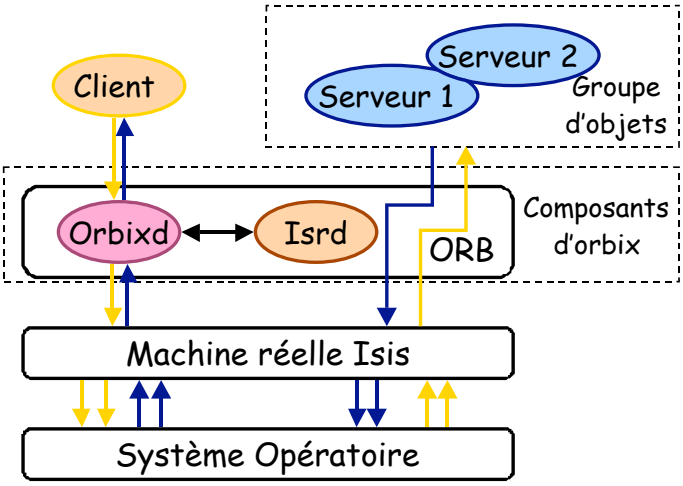


Figure 15. Architecture et interactions dans Orbix+Isis

Comme le montre la Figure 15, l'architecture d'ORBIX+Isis est composée d'un ORB légèrement modifié par rapport à la norme CORBA. Elle inclut un processus de service (daemon) *Orbixd*, couplé avec un système de communication de groupe, en l'occurrence ISIS, qui comprend un autre daemon *Isrd*. (1) *Orbixd* enregistre les informations concernant les serveurs CORBA dans le répertoire d'implémentation (voir la section 1 du chapitre I), leur fait ensuite parvenir les requêtes des clients. (2) Quant au *Isrd*, il enregistre les informations concernant la configuration de la réplication (c'est-à-dire le type de réplication, le nombre des répliques, etc.), identifie les requêtes des clients et gère les communications de groupe ainsi que la diffusion de ces requêtes.

Orbix+Isis envoie des requêtes en utilisant la communication de type point-à-point via orbix, la diffusion étant réalisée au niveau du système de communication de groupe (Isis). Pour pouvoir bénéficier de la réplication des objets, le programmeur de l'application doit choisir entre deux types d'exécution : réplication et flux d'évènements *stream event*. Le rôle du serveur est ainsi fixé lors de l'implémentation par héritage d'une classe abstraite qui fournit les méthodes nécessaires à l'exécution d'un style particulier.

- Le type *réplication* est utilisé pour les applications synchrones. Pour la stratégie de réplication, le contrôle des requêtes se fait de trois façons différentes : (1) La diffusion, où une requête invoque une méthode de l'ensemble des répliques actives du groupe. (2) Le choix du client, où une requête (de lecture seule) invoque une méthode d'une seule réplique active du groupe, choisie par le client. (3) Le coordinateur-cohorte implémente la réplication passive. Dans ce cas, une seule réplique (le primaire) est invoquée et les autres seront mises à jour après retour de la réponse. Les aspects tolérance aux fautes (i.e. le mode de réplication) et performances (i.e. délais de délivrance d'une requête) d'une application Orbix+Isis sont spécifiés dans le répertoire d'ISIS (*ISR*). Ce dernier extériorise ces informations afin de permettre le changement du comportement de l'application sans avoir à recourir ni au changement de son code ni même à sa re-compilation.

- Le type *flux d'évènements* est utilisé pour les applications asynchrones. Il est basé sur le paradigme *publication-souscription*, chaque groupe de répliques étant représenté par un flux d'évènements. Tout serveur voulant faire partie d'un groupe de répliques donné doit s'inscrire auprès de son flot d'évènements. Par ailleurs, tout client voulant envoyer une requête doit impérativement la publier au niveau du flot d'évènements du groupe. L'utilisation de ce type d'exécution découple les clients des serveurs et offre par la suite une persistance aux évènements ; en effet, un client peut en disposer après avoir envoyé une requête, cette dernière pouvant rester au niveau du flot d'évènements. Ce type d'exécution pourrait être utilisé lors d'un recouvrement des répliques défaillantes pour récupérer une copie des requêtes.

Cette approche, offre une transparence totale des mécanismes de tolérance aux fautes aux clients, puisqu'aucune interface des composants de l'architecture de l'ORB n'a été modifiée. De plus, la gestion des requêtes est dynamique, car le passage d'une configuration de réplication active à une réplication passive est réalisé sans recours à la recompilation des applications (client/serveur). En revanche l'utilisation d'un processus de service au niveau de l'ORB, pour rediriger les invocations vers le service de communication de groupe, pénalise cette approche puisqu'elle intègre un nouveau composant à l'intérieur de l'ORB à travers lequel transitent les requêtes. Ceci n'est pas conforme à la norme CORBA.

La qualification de cette approche d'implicite provient de l'utilisation d'un mécanisme d'écoute (daemon) au niveau de l'ORB pour rediriger les requêtes IIOP sortantes, vers un GCS. Dans cette approche, le champ "IOR" est extrait de la requête IIOP pour déterminer le groupe de réplique auquel appartient le serveur invoqué par le client et auquel le GCS doit envoyer l'information.

II.1.2.2. ETERNAL

L'objectif de l'architecture d'ETERNAL [Moser 1999, Narasimhan 2001] est de pallier le problème d'interopérabilité soulevé par l'intégration des services de tolérance aux fautes dans les ORB. En fournissant une couche logicielle entre l'ORB et le système d'exploitation, ETERNAL cache les mécanismes de tolérance aux fautes aux applications et l'ORB reste compatible. Cette approche est basée sur l'interception des messages IIOP qui transitent entre l'ORB et le système d'exploitation pour soumettre les requêtes aux mécanismes de tolérance aux fautes préalablement définis. D'où sa classification d'*approche par interception* [Felber 1998a].

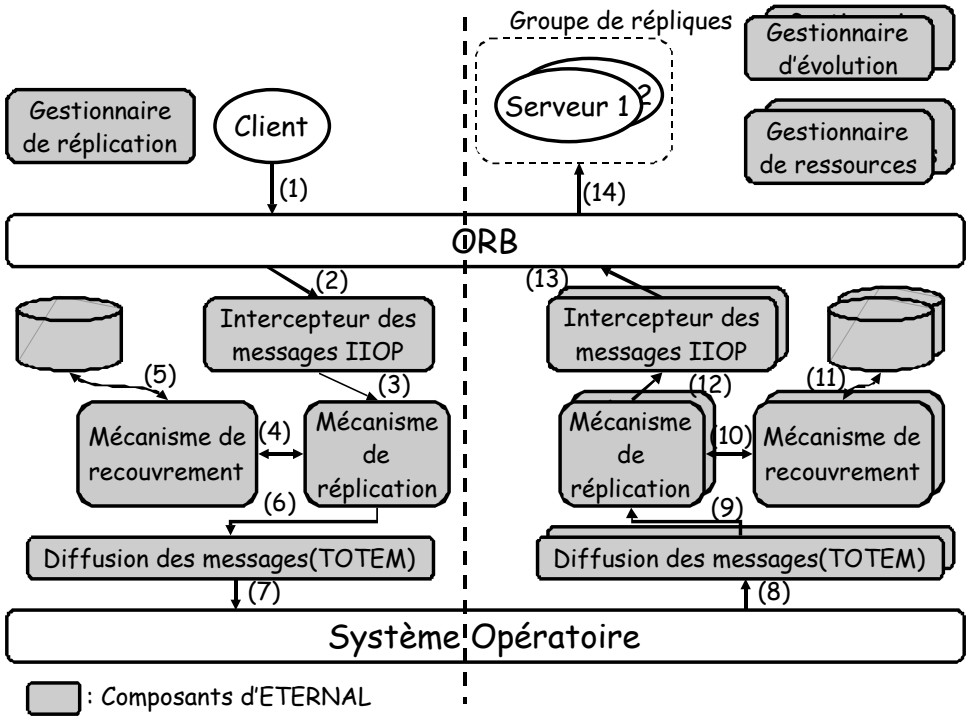


Figure 16. Architecture et Interactions dans ETERNAL

Comme le montre la Figure 16, ETERNAL est composé d'objets qui se trouvent au-dessus de l'ORB et de mécanismes qui se placent entre l'ORB et le système opératoire :

- La partie supérieure renferme : (a) un *gestionnaire de réplication* qui est responsable de la duplication des objets de l'application selon les propriétés définies par l'utilisateur (mode de réplication, intervalle de capture de l'état, périodicité de la détection des défaillances, nombre initial de répliques, nombre minimal de répliques, etc.) et les répartit sur différents sites du réseau ; (b) un *gestionnaire de ressources* qui observe les ressources de l'ensemble du système (i.e. l'application, le système de réplication et le système opératoire) et assure le maintien du nombre minimal de répliques ; (c) un *gestionnaire d'évolution* qui exploite la réplication des objets pour donner la possibilité à l'application distribuée d'être mise à jour. Ces différents gestionnaires offrent des interfaces compatibles avec la spécification de FT-CORBA de l'OMG [OMG 2002b].

- La partie inférieure est composée: (a) *d'un intercepteur*, (b) du *mécanisme de réplication*, (c) du *mécanisme de recouvrement* couplé avec sa base de données associée et (d) du *système de diffusion* de messages, en l'occurrence TOTEM. L'*intercepteur* capture les messages IIOP (contenant les requêtes du client et les réponses du serveur) destinés à la couche de transport TCP/IP, et les redirige vers le mécanisme de réplication. Le mécanisme de réplication joue un double rôle : d'une part, il détermine la référence du groupe destinataire pour les requêtes sortantes (côté client) avant de les délivrer au service de groupe (TOTEM, en l'occurrence) [Moser 1996]; d'autre part, il garde l'information relative aux répliques non défaillantes pour délivrer les requêtes entrantes (côté serveur) à l'ensemble des serveurs actifs. Par

exemple, il livre la requête au serveur primaire dans le cas de la réplication passive et à l'ensemble des répliques dans le cas de la réplication active. Le mécanisme de recouvrement est responsable du maintien d'un journal des messages et de la prise de points de reprises ainsi que du transfert et la mise à jour de l'état. Le système de diffusion des messages assure une communication atomique (i.e. garantit un ordre total pour la livraison des requêtes). Ces mécanismes sont d'un intérêt incontournable pour que les interfaces de la partie supérieure puissent remplir leurs tâches correctement. Par ailleurs, ces mécanismes pourraient être déployés au dessus de l'ORB mais ils ont été placés autrement pour des raisons de transparence et de performance.

Lors de l'initialisation d'un groupe de répliques, le gestionnaire de réplication crée d'une part une référence unique pour chaque serveur, et d'autre part, une référence de groupe pour identifier l'ensemble des répliques appartenant au même groupe. Ces informations sont enregistrées par le mécanisme de réplication. Ainsi, lorsqu'un client envoie une requête à un serveur donné (1), l'intercepteur la redirige vers le mécanisme de réplication (2) ; ce dernier commence par identifier la référence du groupe cible à partir de celle du serveur pour ensuite la délivrer, ainsi que le message IIOP, au service de recouvrement (4). Le service de recouvrement enregistre la requête et élabore un en-tête à ce message (5), renfermant un identificateur unique et la référence du groupe, avant de retourner le nouveau message au mécanisme de réplication. Ce dernier utilise l'en-tête du message IIOP pour demander (6) au service de groupe de diffuser la requête aux différents membres du groupe (7) d'une manière atomique. Une fois arrivée du côté serveur (8), la requête est délivrée par le GCS au mécanisme de réplication (9). À ce niveau, l'en-tête du message est utilisé pour déterminer la réplique à laquelle la requête sera délivrée. Le message est par la suite envoyé au mécanisme de recouvrement (10) pour qu'il élimine la duplication d'un message, l'enregistre puis élimine son entête qui est spécifique à ETERNAL (11). Une fois le message IIOP retourné au mécanisme de réplication, ce dernier consulte le mode de réplication de l'application (active ou passive) ainsi que le serveur auquel il est associé pour laisser passer la requête (i.e. cas de réplication active et serveur primaire pour une réplication passive) (12) ou la bloquer (i.e. cas de serveur secondaire pour une réplication passive). À ce niveau, l'intercepteur d'ETERNAL n'est activé (13) que pour passer les requêtes aux serveurs cibles (14).

En utilisant l'interception des requêtes IIOP, ETERNAL masque au client le fait qu'un service est répliqué. En effet, même si le client envoie une requête vers un serveur défaillant, cette requête sera interceptée au niveau du système opératoire (connexion sur sockets) puis diffusée vers les serveurs opérationnels du groupe. En revanche, côté serveur, cette transparence est partiellement perdue. Comme l'interface du service doit hériter de l'interface `checkpointable`, cette approche est intrusive pour le code du serveur. De plus, l'implémentation des méthodes `set_state` et `get_state` de cette interface sera à la charge du développeur de l'application, qui aura à fournir l'état de l'application, l'état du POA et l'état de l'infrastructure, ce qui ne relève pas forcément de ses compétences. Par ailleurs, comme cette approche utilise des gestionnaires (réplication, évolution et ressources) qui sont des objets (services) CORBA, on pourrait penser qu'elle peut être utilisée sur

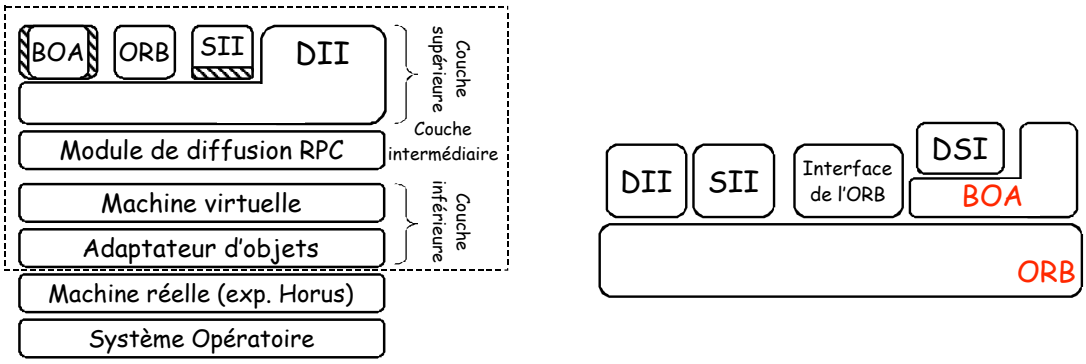
n'importe quel système d'exploitation. En pratique l'utilisation de l'interception des messages IIOP à un niveau très bas, à partir du fichier */proc*, la rend dépendante de certains types de système opératoire, en l'occurrence UNIX.

Comme dans l'approche Orbix+Isis, la qualification de l'approche ETERNAL d'implicite est due à son utilisation implicite de l'IOR pour atteindre l'ensemble des membres d'un groupe d'objets. En effet, un message IIOP est intercepté au niveau du système opératoire, puis analysé par le mécanisme de réplication pour en extraire le champs relatif à l'IOR du serveur cible. À cette IOR correspond la référence du groupe des répliques que ce mécanisme utilise pour envoyer la requête à travers le service de diffusion atomique aux différents serveurs répliqués.

II.1.3. Approches explicites

II.1.3.1. ELECTRA

Basé sur la version 1.2 de la spécification CORBA, ELECTRA [Maffeis 1995] est historiquement le premier ORB qui a visé à introduire la notion de tolérance. Implémenté en C++, cet ORB intègre les fonctionnalités de gestion de groupe et les mécanismes de tolérance aux fautes pour faciliter le développement d'applications distribuées sûres de fonctionnement.



▨: Méthodes d'interface non standards

Figure 17 a. Architecture d'ELECTRA b. Aperçu de l'architecture de CORBA 1.2

Comme le montre la Figure 17.a, l'architecture d'ELECTRA est composée de trois parties :

- La *couche supérieure* représente l'interface de programmation des applications (API). Elle fournit de nouvelles fonctionnalités relatives à la tolérance aux fautes (i.e. gestion d'un groupe de répliques, détection d'une défaillance, etc.).

- La *couche intermédiaire* est responsable de l'appel à distance d'une procédure. Elle joue le même rôle qu'un ORB ; elle est destinée, particulièrement, à des applications procédurales et non pas orientées objets.

- La *couche inférieure* a pour rôle de rendre ELECTRA portable par rapport aux différents systèmes de communication de groupe (i.e. ISIS, HORUS, TOTEM, etc.).

L'interface proposée par la partie supérieure est très proche de celle définie par l'OMG (voir Figure 17.b) à l'exception de trois composants. La première différence réside au niveau du BOA qui est la version non portable du POA introduit dans le paragraphe I.1. Ce composant s'est vu rajouter d'une part des méthodes de gestion de groupes (la création ou la destruction de groupes d'objets CORBA, et l'inscription ou le retrait d'une réplique à un groupe d'objets) et d'autre part, des services de tolérance aux fautes, à savoir des fonctionnalités relatives à l'obtention et la mise à jour de l'état interne des différents membres d'un groupe et à la notification des changements d'appartenance. La seconde différence concerne l'interface de la classe *environment*, qui est responsable de la transmission des exceptions soulevées par le serveur au client. Définie par l'OMG, cette classe se voit ajouter une première méthode qui spécifie le type de la requête (i.e. synchrone, asynchrone ou intermédiaire), puis une seconde qui fixe le nombre de réponses qu'un client reçoit. La dernière différence réside au niveau de la SII. En effet, cette interface est générée par le compilateur IDL d'ELECTRA qui est basé sur l'interface DII. Contrairement à la spécification de l'OMG qui propose à la fois une SII et une DII, ELECTRA considère le DII comme le cœur de l'ORB. Ainsi toute requête est construite dynamiquement en utilisant son type (i.e synchrone etc.) que l'ORB récupère de la classe *environment*. L'empaquetage des données, *marshaling*, n'est pas réalisé au niveau de l'ORB mais au niveau de la DII, donc à l'exécution (d'où un surcoût).

De son côté, la couche inférieure est composée d'une machine virtuelle et d'un adaptateur d'objets. La machine virtuelle offre la même abstraction des mécanismes de diffusion, de gestion de groupes qu'un GCS. La seconde composante transforme les méthodes définies au niveau de la machine virtuelle (gestion de groupe, contrôle des différents types de messages, détection des erreurs, synchronisation par sémaphores ou par compteurs d'événements) vers leurs équivalents dans la plate-forme spécifique (i.e Isis, Horus, etc.).

Avec l'ajout de nouvelles méthodes à l'interface BOA, la création d'un groupe d'objets et son initialisation avec un membre sera à la charge de l'application elle même. En outre, même si l'invocation d'une méthode d'un groupe d'objets est transparente au client, elle est non-interopérable. En effet, dans l'approche ELECTRA, un objet est un groupe d'objets avec un membre unique. Ainsi, le client doit récupérer la référence d'un groupe pour invoquer la méthode d'un serveur. Une fois arrivée au niveau du module RPC, cette requête sera émise vers les différents membres du groupe. Seuls la détection de l'arrêt inopiné d'un membre, le contrôle et la mise à jour de l'état interne de groupe sont transparents par rapport aux clients et aux applications.

Le changement de la sémantique de l'IOR a valu à cette approche sa qualification d'explicite. En effet, dans ELECTRA, l'IOR référence un groupe de répliques et non plus un objet CORBA unique. Ainsi, l'utilisation de l'IOR par un client explicite le groupe d'objets auquel une invocation doit être diffusée par le service RPC.

II.1.3.2. OGS (*Object Group Service*)

Élaboré dans le cadre de travaux visant à munir l'intergiciel CORBA de mécanismes de tolérance aux fautes d'une manière inter-opérable, OGS [Felber 1998b] propose un service de gestion de groupe dont l'interface est mise à la disposition des développeurs d'applications distribuées. Dans cette approche, on n'a apporté aucune modification au niveau de l'API proposée par l'OMG, d'où sa classification dans les approches *service*.

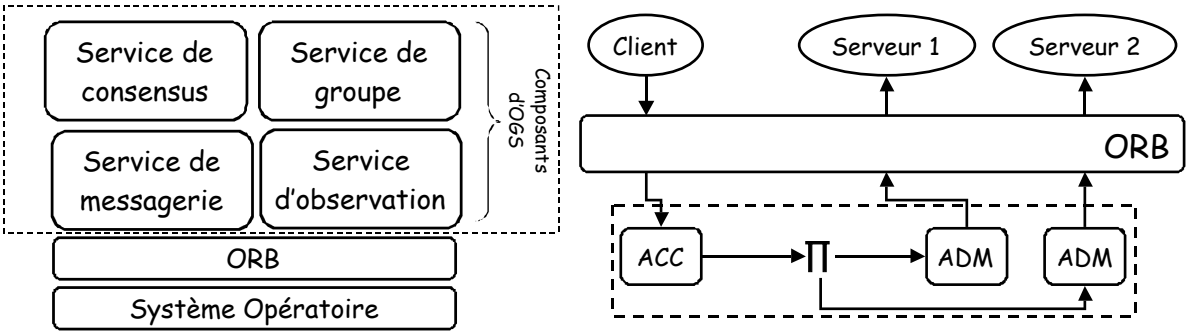


Figure 18. a. Architecture de OGS b. Interactions client/serveur dans OGS

Comme le montre la Figure 18.a, OGS propose quatre services différents :

- Le *service de groupe* est responsable, d'une part, des opérations relatives au cycle de vie d'un groupe d'objets (i.e. la création, l'ajout ou le retrait d'un membre et la destruction d'un groupe) et d'autre part, de diffuser les requêtes vers les membres actifs du groupe.

- Le *service d'observation* fournit une interface composée des méthodes permettant au service de groupe d'observer l'état courant des membres d'un groupe (défaillance, élimination ou encore création d'un serveur).

- Le *service de messagerie* fournit une interface utilisée par les clients pour pouvoir envoyer leurs requêtes asynchrones à un groupe d'objets. En effet, comme l'intergiciel CORBA utilise un modèle d'appel synchrone pour le traitement des requêtes (le client reste bloqué lorsque le serveur est en train de traiter la requête), le service de messagerie ajoute de nouvelles fonctionnalités pour permettre aux clients d'émettre des requêtes non bloquantes et au groupe d'objets de diffuser les requêtes en utilisant des primitives au niveau système.

- Le *service de consensus* permet de résoudre les problèmes relatifs au consensus entre objets distribués avec communications asynchrones [Fischer 1985]. Ce service implémente l'algorithme de Chandra et Toueg [Chandra 1996] pour permettre à des objets distribués de s'accorder sur le nombre des membres actifs dans un groupe.

La Figure 18.b montre les interactions entre client et serveurs en utilisant l'approche service d'OGS. Lorsqu'un serveur est lancé, il commence par créer un *administrateur de groupe* (ADM) qui offre une interface permettant aux serveurs de gérer le cycle de vie du groupe auquel ils appartiennent ou souhaitent appartenir (i.e. insertion, retrait, etc.). Par ailleurs,

lorsqu'un client est initialisé, il crée de son côté un *accesseur de groupe* (ACC). C'est l'interface de l'ACC qui permet au client de pouvoir interagir avec un groupe d'objets. Ainsi, toute requête issue d'un client arrive au niveau de l'*accesseur de groupe*, puis est reprise par le service de messagerie pour être livrée (diffusée) aux différents membres du groupe via leurs *administrateurs de groupe*.

En utilisant cette méthode pour la réplication des applications, le problème de la portabilité de cette approche est théoriquement résolu par définition. En effet, en utilisant des services CORBA, tout système d'exploitation qui supporte ce type d'intergiciel peut être utilisé sans modification aucune au niveau du code des services (applications). En revanche, comme cette approche est basée sur les appels de méthodes définies au niveau des interfaces proposées, la gestion des groupes et la transmission des invocations ne sont pas transparentes. Ceci est dû, d'une part à la demande explicite du serveur pour créer un *administrateur* (création ou ajout d'un membre d'un groupe) et d'autre part, à la demande du client de la création d'un *accesseur* qui jouera le rôle de médiateur pour les requêtes qu'il émet pour le groupe d'objets. Par ailleurs, tout service est amené a implémenter l'interface *Groupable*, qui permet de récupérer et de mettre à jour l'état d'un membre du groupe. Ceci impose, d'une part, au développeur de l'application de spécifier l'état de son application, d'autre part, cette approche pose des problèmes d'interopérabilité puisqu'on peut avoir des formats de captures d'état qui ne sont pas compatibles entre différentes implémentations.

L'utilisation d'un *accesseur* de groupe a valu à OGS sa qualification d'approche *explicite*. En effet, un client adresse explicitement sa requête à l'accesseur du groupe pour qu'elle soit diffusée vers l'ensemble des répliques. Dans cette approche, l'IOR ne perd pas son sens, celui de l'identification d'un objet CORBA unique. Le client n'utilise plus l'identificateur du serveur pour accéder aux services qu'il propose. Néanmoins, il utilise l'identificateur de l'*accesseur* qui peut le récupérer aussi bien lors sa création (cas où l'*accesseur* est créé par le client) qu'en utilisant le service de nommage, si l'*accesseur* est créé par un membre tiers.

II.1.4. Analyse des approches

Dans les approches implicites, qu'elles fonctionnent par interception ou par intégration, les requêtes sortantes sont déroutées par défaut, soit par le mécanisme d'interception des requêtes IIOP au niveau du système opératoire (i.e. ETERNAL) ou en utilisant un mécanisme d'écoute "*daemon*" au niveau de l'ORB (i.e. ORBIX+ISIS), vers un service de communication de groupe qui réalise par la suite la diffusion de la requête en question. Dans les deux approches, le client utilise la référence d'une réplique pour que sa requête soit diffusée aux différents membres du groupe.

Dans les approches explicites, qu'elles soient de type service ou par intégration, les requêtes sortantes sont envoyées non plus à un serveur d'application mais à un mandataire (notion d'accesseur dans OGS) ou encore à une référence d'un groupe d'objets (i.e. ELECTRA). Si, pour OGS, le sens d'un IOR n'a pas été modifié car il identifie un accesseur, il a été modifié dans ELECTRA, puisqu'un IOR représente un groupe d'objets et non plus un objet (un objet

sera identifié comme un groupe d'objets renfermant un objet unique). Avec ELECTRA et OGS, l'IOR utilisé n'est pas celui d'une réplique cible mais il représente l'interface d'un groupe de répliques.

Dans la classification utilisée antérieurement (i. e. interception, intégration et par service) les approches ORBIX+ISIS et ELECTRA sont regroupées dans la même classe alors qu'elles ont des caractéristiques différentes en termes de transparence pour le client.

En ajoutant une nouvelle dimension (Tableau 3) ayant trait à l'aspect implicite ou explicite de l'approche, nous pouvons distinguer entre les approches ORBIX+ISIS et ELECTRA. Par ailleurs, cette nouvelle classification permet de prévoir les approches par service implicite et des approches par interception explicite. Dans la première, par exemple, la gestion de la tolérance aux fautes peut être gérée par la plate-forme sans que l'utilisateur ne s'en aperçoive. Dans la seconde, nous pouvons trouver des systèmes dans lesquels le mécanisme d'interception est choisi par l'application.

	Interception	Intégration	Service
Implicite	ETERNAL	ORBIX + ISIS	
Explicite		ELECTRA	OGS

Tableau 3. Nouvelle classification

II.2. LES APPROCHES REFLEXIVES

Comme nous l'avons présenté dans la section 3 du chapitre I, un système réflexif est défini par deux niveaux : un *niveau de base*, qui représente le service que doit rendre l'application, et un *méta-niveau*, qui représente des fonctionnalités qui lui sont orthogonales (par exemple, la tolérance aux fautes). Nous allons, dans les sections suivantes, présenter certains travaux qui utilisent la réflexivité pour rendre tolérantes aux fautes des applications distribuées à base d'un intergiciel CORBA. Ces travaux seront classés selon l'intégration des mécanismes fonctionnels et non-fonctionnels par rapport à la couche intergicielle et à la couche applicative.

II.2.1. FRIENDS

L'objectif de FRIENDS [Fabre 1998] (*Flexible and Reusable Implementation Environnement for Dependable Systems*) est de fournir un canevas orienté-objet qui permet de munir une application distribuée de mécanismes de tolérance aux fautes (par exemple les différents types de réplication et la gestion de communications de groupe) d'une manière non intrusive. Comme le montre la Figure 19, cette approche est très proche du concept de la *tour*

réflexive infinie de Smith [Smith 1984]. En effet, les clients communiquent avec leurs serveurs via des mandataires (i.e. méta souche et méta objet) ; toute requête reçue par un mandataire (côté client) est déroutée vers son méta objet ; puisque tout méta-objet est un objet, il peut lui aussi avoir un méta-objet et ainsi de suite. L'arrêt de cette infinité potentielle est exprimé explicitement au niveau d'un objet, en ne lui définissant pas de méta-objet. Contrairement à la théorie de Smith, les différents niveaux de FRIENDS ne traitent pas la requête simultanément. On a tiré profit de la succession du traitement des requêtes en implémentant différents mécanismes dans différents niveaux. Ces caractéristiques améliorent la composition du système final, ce qui peut être exploité à des fins de modifications dynamiques des mécanismes.

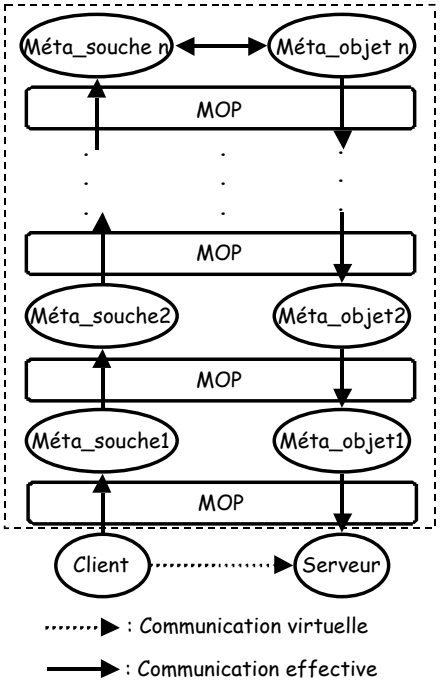


Figure 19. Architecture de FRIENDS

Dans la version 2[7] [Ruiz-García 2003] du protocole à méta-objets FRIENDS, la communication est assurée par l'intergiciel CORBA. Ainsi, tous les objets et méta-objets sont définis par leur interface IDL. Comme le montre la Figure 20, l'architecture de FRIENDS v2 est composée :

- d'un serveur implémentant l'interface IDL du service qu'il fournit : le code du serveur est instrumenté par le compilateur ouvert OpenC++v2 [Chiba 1995] en lui ajoutant une nouvelle interface (`ReifiedObject`) pour qu'il puisse être contrôlé par le méta-niveau;

[7] FRIENDS v1 [Fabre 1998] est un protocole à méta-objet à l'exécution. Il utilise le compilateur ouvert OpenC++v1 [Chiba 1993] pour mettre en œuvre les mécanismes de tolérance aux fautes. Il n'utilise pas un ORB, mais la distribution repose sur des protocoles de communication de groupe fournis par le logiciel xAMp[Verissimo 1990].

- d'un client, qui demande les services proposés par le serveur : contrairement au code du serveur, celui du client n'est pas instrumenté ; la modification touche la souche qu'il utilise, afin de *réifier* les invocations sortantes;

- d'un mandataire (stub), spécifique au MOP FRIENDS : en plus de son rôle d'intermédiaire pour un serveur, il offre une interface appelée `ReifiedStub`, qui est destinée aux interactions avec sa méta-stub;

- d'un méta-mandataire (méta-stub), il implémente l'interface `Metastub`, via laquelle il communique avec le stub (i.e. les invocations réifiées lui arrivent). Par ailleurs, il est responsable du fonctionnement du stub et inter-agit avec le méta-objet;

- d'un méta-objet, qui implémente l'interface *Metaobject*, à travers laquelle arrivent les requêtes réifiées issues du Méta-Stub. Il contrôle le comportement et l'état du serveur grâce au MOP à travers l'interface *Reifiedobject*.

- et, d'une usine à méta-objets, qui crée les méta-objets (respectivement méta-stubs) à la demande du serveur (respectivement du *stub*). Par ailleurs, cette usine initialise les liens entre les différentes entités du méta-niveau pour qu'ils puissent inter-agir.

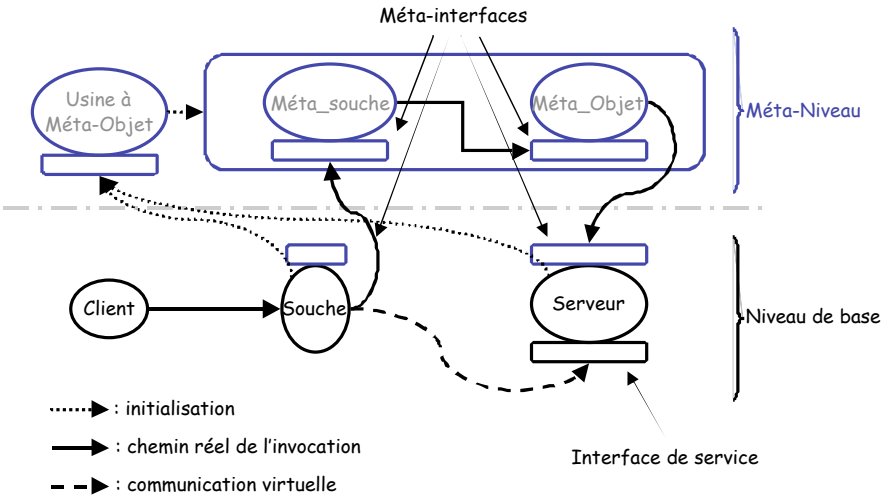


Figure 20. Cas de réflexivité de deux niveaux dans FRIENDS v2

Comme le montre la Figure 21, la mise en œuvre d'une architecture tolérant les fautes peut être réalisée en mettant en jeu deux méta-niveaux [Killijian 2000]. Le premier est responsable de la mise en œuvre des mécanismes de réplication (par exemple : communication de groupe) à travers le méta-objet (MO1) et la méta-souche (MS1). Le second a pour charge de fournir un support de communication fiable (par exemple : l'ajout d'un CRC) en modifiant les requêtes au niveau du méta-méta-objet (MO2) et de la méta-méta-souche (MS2). En effet, toute requête envoyée par le client est réifiée par la souche instrumentée S-S vers sa méta-souche MS1 qui traite la requête du point de vue mécanisme de réplication. Après ce premier traitement, cette requête est réifiée une seconde fois par S-MO1 au MS2 qui lui applique son codage et l'envoi au MO2 via S-MO2. Le méta-objet MO2 décode la requête et la livre au

méta-objet MO1 qui applique le mécanisme de réplication puis la délivre au serveur cible. La souche S-S (respectivement S-MO1 et S-MO2) représente l'interface du serveur (respectivement de MO1 et MO2).

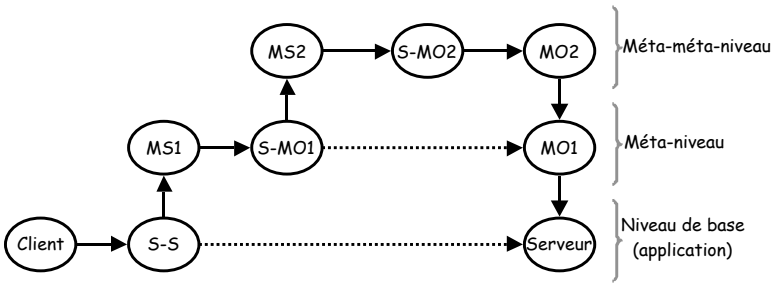


Figure 21. Mise en œuvre de la tolérance aux fautes avec FRIENDS v2

Cette approche met en œuvre des systèmes réflexifs dont le niveau de base et le méta-niveau sont définis dans la couche applicative. Les niveaux sous jacents, intermédiaire ou système d'exploitation, ne sont pas impliqués dans les mécanismes réflexifs puisque, seul le code du serveur d'application est instrumenté. Cette instrumentation, qui fournit tout l'éventail des mécanismes réflexifs, est réalisée d'une manière automatique par OpenC++, ce qui diminue la probabilité d'introduction de fautes. Aussi, cette approche fournit une interface unique pour assurer les interactions entre niveau de base et méta-niveau ainsi qu'entre les différents méta-niveaux. Ceci rend cette approche indépendante de l'application. Néanmoins, elle est quelque peu intrusive puisqu'on effectue des modifications au niveau du code de l'application.

II.2.2. OpenORB

L'objectif du projet OpenORB [Clarke 2001] est de fournir un intergiciel évolutif et configurable. Pour répondre à ces exigences, OpenORB a été conçu de manière à être modulaire : il est constitué d'un ensemble de composants aux interfaces clairement définies et dont les dépendances (interactions) apparaissent aussi clairement. Chaque composant peut être remplacé.

En outre, OpenORB utilise la réflexivité pour renforcer les propriétés architecturales désirées[8] en modifiant les interactions entre les différents composants selon le contexte fonctionnel. Ainsi, avec son potentiel d'observation et d'action, la réflexivité modifie les règles d'interactions entre les composants formant l'intergiciel afin de changer sa structure et son comportement ; néanmoins, cette reconfiguration est contrôlée par les contraintes spécifiées au niveau de la plate-forme à composants dans le but de préserver la stabilité du service rendu.

[8] Ces propriétés peuvent être fonctionnelles (par exemple, le découpage du service entre les différents composants) et non fonctionnelles (par exemple, la modification ou la performance de l'assemblage des composants).

OpenORB est structuré sous forme d'un ensemble configurable de plate-formes à composants. La réflexivité est utilisée pour découvrir la structure et le comportement aussi bien des plate-formes que des composants qui y sont chargés, et pour effectuer des changements à l'exécution.

La Figure 22 représente une implémentation de la partie fonctionnelle d'OpenORB (i.e. niveau de base), conformément à la norme CORBA.

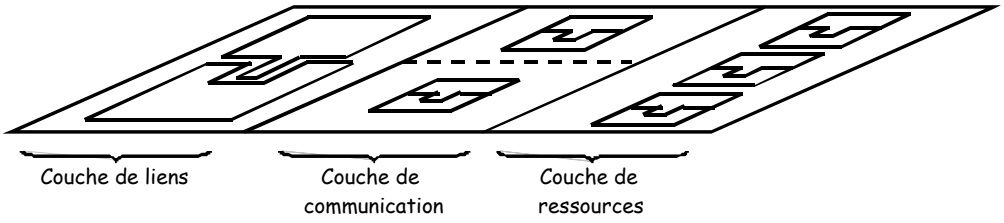


Figure 22. L'implémentation *OpenORB* de la norme CORBA

Cette implémentation est composée d'un ensemble de trois couches de plate-formes à composants qui représentent des canevas conceptuels (*design pattern*) :

- la *couche de liens* : c'est une structure renfermant plusieurs types lieurs. Chaque type de lieur implémente une forme d'interaction, par exemple l'appel de méthodes à distance *Remote Method Invocation* (*RMI*) producteur-consommateur, file de messages et communication de groupe. Ainsi, le rôle de cette structure est de séparer l'aspect communication et coordination de l'aspect traitement effectif pour simplifier le développement des applications et promouvoir la réutilisation des mécanismes d'interactions.

- la *couche de communication* : elle contient la structure de composants en charge des protocoles de communication. Avec cette structure, le gestionnaire de reconfiguration maintient l'information concernant la pile de protocoles qui peut être adaptée depuis le méta-niveau. La structure des composants assure le maintien de l'intégrité de la structure de pile durant la reconfiguration. Cette couche intègre une structure de composants qui fournit un protocole de communication multipoint pouvant supporter un service de communications de groupe.

- la *couche de ressources* : elle est composée de plusieurs structures de composants pour la gestion des files d'attentes, du transport et des brins d'exécution. Cette couche permet l'adaptation de la gestion des ressources à une application donnée. Par exemple, dans une application de transmission multimédia, la gestion des brins permet l'installation dynamique d'un composant « Ordonnanceur » qui assure la synchronisation entre l'image et le son à la réception.

La Figure 23 illustre l'aspect réflexif d'OpenORB.

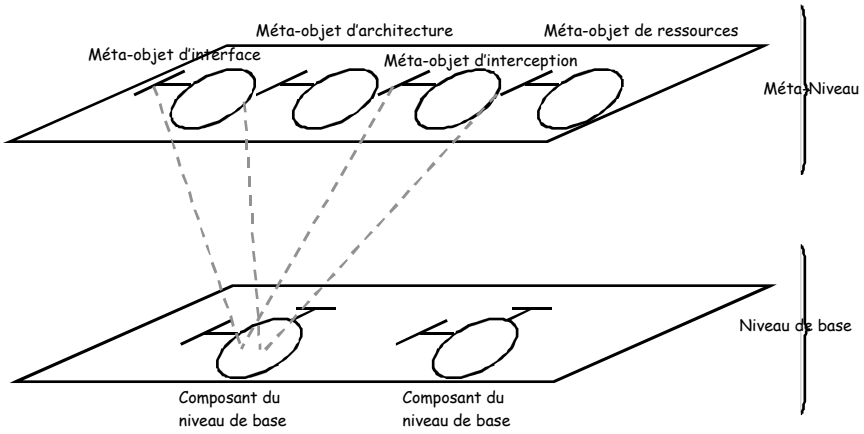


Figure 23. L'aspect réflexif dans OpenORB

Les composants du niveau de base font partie des plate-formes à composants, présentées précédemment. Ils sont développés à l'aide de la technologie *OpenCOM* [Clarke 2001], qui est une extension de l'architecture du *Component Object Model* (COM) de Microsoft. OpenCOM munit les composants COM d'interfaces non fonctionnelles (i.e. méta-interfaces) qui assurent la communication entre le niveau de base et le méta-niveau. Par ailleurs, le méta-niveau est formé de plusieurs méta-modèles différents couvrant aussi bien l'aspect structurel que l'aspect comportemental ; l'utilisation de différents méta-modèles a pour objectif de séparer les fonctionnalités et de réduire la complexité des méta-interfaces en les différenciant. L'aspect structurel concerne le contenu et la structure d'un composant. Quant à l'aspect comportemental, il différencie les actions évoluant à l'intérieur du système des ressources mises à la disposition de l'accomplissement de ces actions. Ainsi, on dénombre quatre méta-modèles différents qui sont les suivants :

- le méta-modèle d'interface : il permet l'accès à la représentation externe d'un composant en termes d'interfaces fournies et requises (i.e. introspection). Ce méta-modèle ne permet pas l'accès à l'implémentation d'une interface, conformément à l'approche par composant qui découple l'interface de son implémentation.

- le méta-modèle d'architecture : il permet l'accès, en lecture et en écriture, à l'implémentation d'un composant, c'est-à-dire qu'il offre les mécanismes d'introspection et d'intercession structurelle. L'accès à l'implémentation concerne le graphe des composants et l'ensemble des contraintes architecturales (cf. plate-forme à composants). Le concept de graphe des composants est représenté par un ensemble de sous-composants (plus précisément d'interfaces) connectés entre eux. Normalement, cette structure est cachée à l'utilisateur des composants ; cependant, le méta-modèle architectural peut être utilisé pour découvrir et adapter cette structure à l'exécution.

- le méta-modèle d'interception : il est associé aux interfaces des composants, ces méta-modèles réifient toutes les interactions qui y transitent ; ils permettent aussi l'ajout dynamique d'intercepteurs ce qui peut introduire de nouveaux pré/post traitements aux requêtes.

- le méta-modèle de ressources : il permet l'accès aux ressources sous-jacentes ainsi qu'à leurs gestionnaires. Basé sur l'abstraction des ressources et des tâches, ce méta-modèle peut agir sur les tâches élémentaires pour les réorganiser.

Pour munir les applications de mécanismes de tolérance aux fautes, on a défini une plate-forme de communication de groupe appelé GOORB *Group support for Open ORB* intégrée à l'intergiciel OpenORB [Saikoski 2003]. Cette plate-forme est composée d'un service de lien de groupe *Group Binding Service*, d'un service de configuration *Configuration Group Service* et d'un service de reconfiguration *Reconfiguration Service*.

Le service de lien de groupe est situé au niveau de la couche de liens (voir Figure 22). Il peut encapsuler des fonctions de communication, par exemple *IP multicast*, ou les améliorer en ajoutant des composants de tolérance aux fautes, par exemple *Scalable Reliable Multicast, SRM* [Floyd 1997]. Une instance de ce service est créée en utilisant une « usine de groupe ». Cette dernière peut créer un groupe avec des membres issus de composants d'un même ou de différents types.

Le service de configuration est responsable de la configuration, de la création, de la destruction et du contrôle des instances de groupes. Il est responsable de l'initialisation des usines de groupes et du gestionnaire de réplication. Les premières permettent l'initialisation d'instances de groupes, alors que le second se charge de l'ajout, de l'élimination et du recensement des membres lors de l'exécution de la plate-forme.

Le service de reconfiguration est basé sur le méta-modèle d'interface et le méta-modèle d'architecture. Il se charge de la reconfiguration d'une instance de groupe lors de l'exécution. En effet, le méta-modèle d'interface permet de découvrir les interfaces externes offertes par le groupe. Ces interfaces sont relatives aux groupes ouverts[9] qui peuvent être utilisés pour la réplication des services. Le méta-modèle d'architecture est, par ailleurs, utilisé pour adapter l'implémentation du service de communication de groupe à l'exécution. Comme l'implémentation de l'intergiciel Open ORB est basée sur *Open COM*, l'ajout ou le retrait d'un composant du graphe des composants (l'implémentation du service) correspond à l'ajout ou au retrait d'un membre du groupe des répliques. Aussi, l'action sur les arcs du graphe des composants affecte la communication entre les différents membres.

II.2.3. Dynamic TAO

DynamicTAO [Roman 1999] est un intergiciel réflexif qui fournit une interface de programmation (*API*) conforme à la norme CORBA. Il est basé sur l'intergiciel TAO (*The Adaptative common environment Orb*) [Schmidt 1999] qui utilise les patrons de conceptions orientés objets pour séparer les différents aspects des mécanismes internes de l'ORB afin

[9] Les groupes ouverts possèdent des interfaces externes visibles qui permettent aux composants externes d'interagir avec le groupe (GOORB_SRDS).

d'être portable, flexible, extensible et configurable. DynamicTAO apporte à l'infrastructure de *TAO* la possibilité d'inspection et la reconfiguration des stratégies (i.e. la politique de concurrence, le démultiplexage des requêtes, l'ordonnancement et la gestion des connexions) à l'exécution. Ces capacités de reconfiguration sont très utiles pour l'adaptation des systèmes sujets à des variations de ressources :

- dans l'espace, par exemple, avoir un espace mémoire "RAM, disque" à capacité différente et un support de communication "ATM, ETHERNET" avec une bande passante différente,

- dans le temps, par exemple, variation de la disponibilité de la CPU, de la mémoire ou du support de communication.

La Figure 24 présente la partie ajoutée à l'intergiciel TAO qui lui permet d'avoir une structure plus flexible, en particulier lors de son exécution

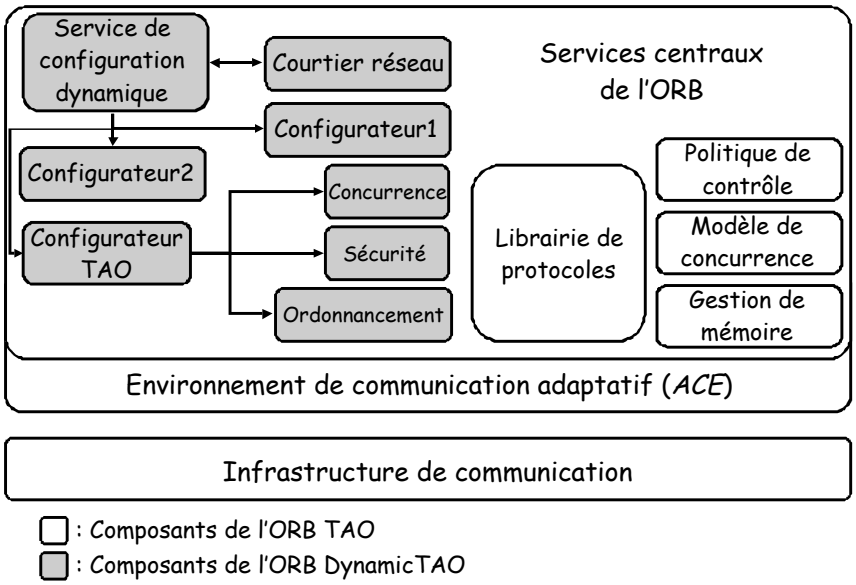


Figure 24. Architecture de l'intergiciel DynamicTAO

À la base, TAO utilise l'environnement de communication adaptative (*Adaptative Communication Environment*, ACE[10]) pour avoir une implémentation de la norme CORBA facilement portable et destinée à des applications temps-réel grâce aux notions de qualité de services introduites par cette plate-forme, comme la gestion de la bande passante et la fixation des délais de transmission. Par ailleurs, il implémente les services centraux de l'ORB CORBA tout en proposant une flexibilité lors de l'initialisation de l'ORB. En effet, TAO permet de choisir à l'initialisation le protocole qu'utilisent les messages lors de leur transmission parmi ceux qui sont disponibles dans la bibliothèque de protocoles. Par exemple, l'ORB peut charger

[10] C'est une plate-forme orientée objets qui implémente plusieurs canevas de conceptions pour des systèmes à communications concurrentes. Elle offre un ensemble d'encapsulateurs d'interfaces à objets (i.e. ils encapsulent les interfaces de l'API C d'un système d'exploitation dans de nouvelles interfaces orientées objets C++).

le protocole IIOP pour des messages GIOP sur TCP, ou alors il peut charger le protocole ESIOP pour les messages temps-réel IOP sur UDP.

Dynamic TAO introduit les composants suivants :

- le courtier réseau (*Network Broker*) qui est une interface à travers laquelle l'intergiciel collecte les requêtes de reconfiguration et d'introspection.

- le service de configuration dynamique (*Dynamic Service configurator*) qui offre une interface permettant de modifier dynamiquement les composants configurables et les implémentations, à savoir l'ORB et les serveurs.

Comme il a été évoqué dans la section 3 du chapitre I, un système réflexif donne la possibilité au programme d'accéder à sa définition et à ses règles d'évaluation et de les altérer via des interfaces particulières. L'intergiciel DynamicTAO s'inscrit dans ce cadre en considérant une requête comme un *programme* et l'implémentation de l'intergiciel comme un *évaluateur*. Ainsi, pour permettre le changement de son implémentation, l'intergiciel *DynamicTAO* offre une méta-interface appelée « interface du service de configuration dynamique ». Toute requête à destination de cette interface peut modifier l'évaluation des prochaines requêtes qui transitent par l'intergiciel. Afin de préserver la stabilité de l'intergiciel, lors de reconfigurations dynamiques, DynamicTAO utilise le patron de conception MEMENTO [Gamma 1997] ; ce patron de conception permet d'extérioriser et d'enregistrer l'état interne d'un objet.

À la manière d'OpenORB, DynamicTAO possède des capacités d'auto-adaptation. Le chargement dynamique d'un composant peut être utilisé pour améliorer la disponibilité d'un service. Par exemple, lorsqu'un système informatique lié à un réseau de type *Ethernet* bascule sur le réseau sans fil, pour des raisons de mobilité, DynamicTAO assure le chargement du composant responsable de l'adaptation du service de transport pour pallier la rupture du service.

Contrairement à OpenORB, le comportemnt de DynamicTAO peut être modifié par l'application qui l'utilise. On peut par exemple utiliser le chargement dynamique d'un composant pour modifier le service de sécurité ou le service de monitoring [Kon 2000]. Dans ce cas, l'utilisateur adresse ses requêtes au configurateur dynamique afin de changer, par exemple, la méthode de chiffrement des messages ou encore la fréquence de stockage des points de reprises.

II.2.4. Classification et analyse des approches réflexives

Nous introduisons une notation concise afin de classifier les systèmes réflexifs à base d'intergiciels. Chaque classe est repérée par un couple <x,y> où la variable « x » désigne l'emplacement du niveau de base et la variable « y » l'emplacement du méta-niveau, comme le montre le Tableau 4. Par exemple, le couple <A,I> représente les applications réflexives dont les mécanismes non-fonctionnels sont intégrés au niveau de l'intergiciel.

Méta-Niveau / Niveau de base	Application	Intergiciel
Application	*<A,A>*	*<A,I>*
Intergiciel	*<I,A>*	*<I,I>*

Tableau 4. Classification générique des systèmes réflexifs

Chaque classe du Tableau 4 présente un potentiel différent pour munir les applications, clients et serveurs, de mécanismes de tolérance aux fautes par réplication :

La *classe <A,A>* est très appropriée pour les trois types de réplication des services au niveau de la couche applicative. En effet, le méta-niveau peut observer et contrôler aussi bien la structure que le comportement de l'application de base via un protocole à méta-objets. Dans le cas de la réplication active, le protocole à méta-objets contrôle le comportement d'un client en déroutant ses requêtes vers un GCS afin de les diffuser aux différents membres d'un groupe de réplique. Dans le cas de la réplication semi-active, les requêtes seront redirigées vers le méta-primaire qui s'occupe d'envoyer les *mini-checkpoints*. Par ailleurs, la récupération de l'état d'un serveur et la mise à jour de l'état d'une réplique sont utilisées pour mettre en œuvre la réplication passive. Comme l'état des couches basses est cependant inaccessible pour cette approche, ceci réduit l'efficacité de sa mise en œuvre.

La *classe<A,I>* est capable de fournir les mêmes performances que la *classe<A,A>*. Néanmoins, cette potentialité est tributaire du moyen utilisé pour observer et contrôler l'application à partir de l'ORB (c'est-à-dire le protocole à méta-objets entre la couche applicative et celle de l'intergiciel). Par ailleurs, cette classe apporte une meilleure transparence par rapport à la mise en œuvre de la tolérance aux fautes par sa localisation, à l'intérieur de l'ORB. Nous évaluons les capacités de cette classe à fournir des mécanismes de tolérance aux fautes à des applications utilisant l'intergiciel CORBA dans le troisième chapitre.

La *classe<I,A>* est appropriée pour la mise en œuvre de la réplication active et semi-active. En effet, les applications peuvent contrôler et modifier les flots des messages qui transitent au niveau de l'intergiciel en modifiant les composants utilisés pour implémenter l'intergiciel. Ainsi, on peut utiliser des composants fournissant les mécanismes de réplication active et semi-active. Par contre, cette classe n'offre pas la transparence de la mise en œuvre des mécanismes de réplication puisqu'ils sont guidés à partir de la couche applicative. Par ailleurs, cette approche n'est pas appropriée pour la mise en œuvre de la réplication passive car les applications du méta-niveau n'ont pas d'accès à l'état des serveurs déployés.

La *classe <I,I>* est très appropriée pour munir une application de la réplication active et semi-active. En effet, un intergiciel qui appartient à cette classe peut auto contrôler : (1) l'envoi des

requêtes (2) et le retour des réponses. Dans le premier cas, l'intergiciel peut assurer une diffusion atomique, choisir les serveurs qui vont recevoir la requête et mémoriser les requêtes qui ont été envoyées. Dans le second cas, l'intergiciel peut inhiber le retour de la réponse d'un serveur, ré-envoyer la requête et accéder aux valeurs de retour. Comme le méta-niveau n'a accès qu'à l'état et au comportement de l'intergiciel, la récupération de l'état de l'application n'est pas accessible. Par conséquent, les approches de cette classe ne peuvent pas être appliquées pour la mise en œuvre de la réplication passive.

Le Tableau 5 résume ces différentes capacités. Les signes "++", "+" et "-" désignent, respectivement, une classe "très appropriée", "appropriée" et "non appropriée" pour un type de réplication donnée.

Classe Réflexive / Réplication	<A,A>	<I,I>	<I,A>	<A,I>
Active	++	++	+	++
Passive	++	-	-	++
Semi-Active	++	++	+	++

Tableau 5. Evaluation des classes réflexives par rapport à la réplication des services

Les différents projets présentés dans ce chapitre, FRIENDS, OpenORB et DynamicTAO, peuvent être repérés par la notation présentée ci-dessus. FRIENDS appartient à la classe <A,A>, OpenORB à la classe <I,I> et DynamicTAO à la classe <I,A>. Bien que tous les trois puissent être qualifiés de réflexifs, il existe une différence fondamentale entre d'une part FRIENDS et d'autre part DynamicTAO et OpenORB. Alors qu'avec le premier, *les éléments réflexifs* se trouvent au sein de l'application, avec les seconds, ils se trouvent dans l'intergiciel. Ainsi on peut qualifier ces deux derniers d'intergiciels réflexifs alors qu'avec le premier c'est l'application qui est rendue réflexive. Une autre différence importante existe entre ces systèmes ; en effet, *les mécanismes réflexifs* (le métaniveau) résident soit au sein de l'intergiciel (cas d'OpenORB), soit au sein de l'application (cas de DynamicTAO et de FRIENDS).

L'architecture DAISY que nous présentons dans le chapitre suivant est repérée par le couple <A,I>. En effet, on montrera que DAISY peut être vue comme rendant *réflexive l'application* (éléments de base au niveau applicatif) et plaçant *les mécanismes réflexifs au niveau intergiciel*.

II.3. LA NORME FT-CORBA

Suite aux différents travaux de recherches qui s'articulent autour de la tolérance aux fautes dans les applications distribuées utilisant des intergiciels, l'OMG s'est associé aux principaux acteurs de ces recherches pour définir la norme *Fault Tolerant CORBA* (*FT-CORBA*) au cours de l'année 1999 [OMG 2002b]. Les objectifs de FT-CORBA se présentent comme suit :

- munir les applications d'un support robuste, c'est-à-dire impliquant la non-existence d'un point singulier de défaillance ;
- pourvoir à la réplication des objets, en fournissant une certaine flexibilité au niveau du nombre de répliques et de leurs emplacements
- offrir plusieurs types de stratégies pour la tolérance aux fautes, comme la réplication passive, la réplication active ;
- permettre le contrôle des stratégies de tolérance aux fautes par l'infrastructure ou par l'application ;
- fournir des services de détection, de notification et d'analyse des fautes aux objets répliqués ;
- permettre aux clients légataires, qui sont développées selon la norme CORBA standard, de pouvoir bénéficier des systèmes tolérants les fautes qui répondent à la norme FT-CORBA.

II.3.1. Vue générale de la norme FT-CORBA

Une application distribuée peut renfermer plusieurs composants qui collaborent entre eux et qui ont été développés par des équipes différentes. Gérer la tolérance aux fautes de telles applications comme celle d'une entité unique est inapproprié. Par conséquent, dans la spécification de FT-CORBA, on définit des domaines de tolérance aux fautes. Un domaine de tolérance aux fautes assure un mode de réplication et une politique de sécurité pour les serveurs d'applications d'une entreprise d'une façon uniforme et cohérente. Chaque domaine peut renfermer un ou plusieurs groupes de répliques qui sont hébergés sur une ou plusieurs machines du site.

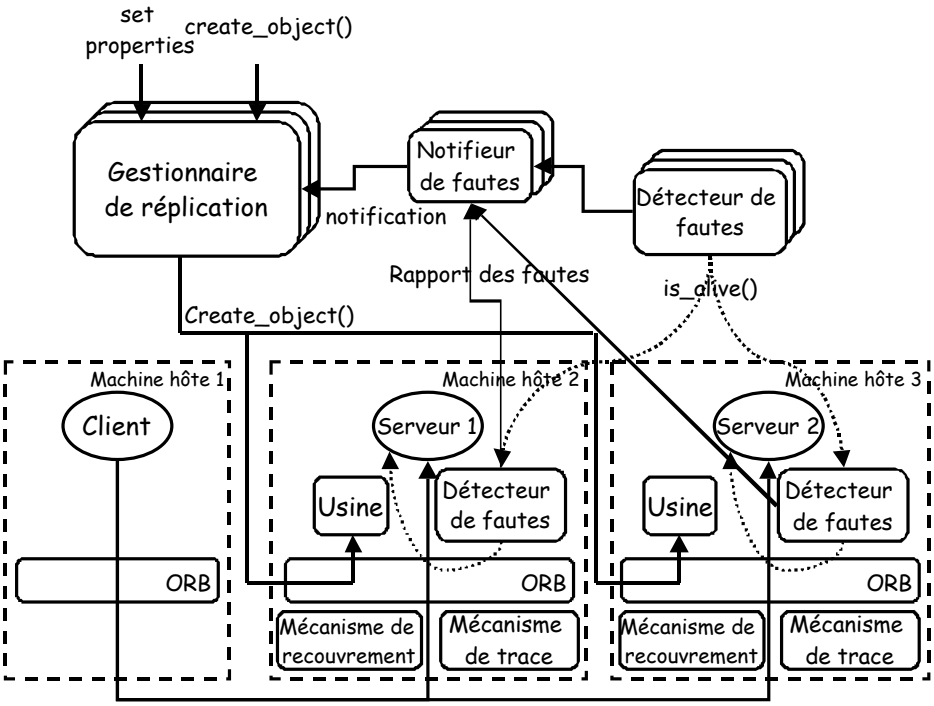


Figure 25. Architecture d'un groupe de répliques selon FT-CORBA

Comme le montre la Figure 25, le groupe de répliques renfermant les serveurs d'applications *Serveur 1* et *Serveur 2* est composé d'une part de trois services répliqués, à savoir le service de *gestion des répliques*, de *détection des fautes* et de *notification des fautes*, et d'autre part des services non répliqués qui sont spécifiques à chaque machine hôte, à savoir l'*usine* à objets et le *détecteur de fautes*. Par ailleurs, un groupe de répliques est composé d'un *mécanisme de recouvrement* et d'un mécanisme de *trace*.

- Le service de gestion des répliques offre (1) le contrôle des groupes d'objet en créant l'identificateur du groupe, en ajoutant ou éliminant un membre du groupe et en choisissant la machine hôte sur laquelle une réplique peut s'exécuter. Il offre aussi (2) le contrôle des propriétés en choisissant le style de réplication, d'adhésion, de consistance, le nombre initial de répliques, etc. lors de la création d'un groupe d'objets. Il permet aussi de changer ses propriétés au cours de son existence. La norme FT-CORBA stipule que le contrôle de la tolérance aux fautes peut être géré par l'infrastructure elle même ou encore depuis l'application. Dans le premier cas, le gestionnaire des répliques demande la création d'un groupe ainsi que de ses membres auprès de l'usine à objets appropriée ; dans le second cas, la création d'un groupe d'objets ainsi que de ses membres se fait par l'intermédiaire de l'usine générique qui est intégrée dans le gestionnaire de répliques.

- Le notifieur de fautes analyse l'ensemble des fautes rapportées par les différents détecteurs du domaine en les filtrant puis en résumant l'ensemble des fautes en un rapport. Toute entité du domaine de tolérance aux fautes, c'est-à-dire une application enregistrée auprès du notifieur ainsi que le contrôleur de répliques, reçoit une copie de

ce rapport pour pouvoir mettre en œuvre leurs stratégies de recouvrement en cas de défaillance.

- Le détecteur de fautes a pour rôle de relever la défaillance d'une réplique. Parmi les deux modèles possibles pour la détection des fautes *push* et *pull*[11], la norme FT-CORBA ne retient que *Pull*. Par ailleurs, les détecteurs de fautes qui résident dans chaque machine ne sont pas répliqués et ne sont pas partageables entre les domaines de tolérance aux fautes. Toute faute relevée au niveau du système opératoire ainsi que la défaillance d'une application est enregistrée puis rapportée par le détecteur au notifieur de fautes (cf. paragraphe précédent).

- Le mécanisme de trace invoque périodiquement l'opération `get_state()` de l'interface `checkpointable`, que chaque application doit implémenter, pour obtenir l'état de la réplique et l'enregistrer dans le fichier de trace. Par ailleurs, le mécanisme de trace stocke dans ce fichier les messages adressés aux répliques qui se trouvent sur la même machine hôte. Lors de la défaillance d'un serveur, primaire par exemple, le mécanisme de recouvrement invoque la méthode `set_state()` de l'interface `checkpointable` du nouveau primaire pour mettre à jour son état en utilisant les données récupérées du fichier de trace. Ces deux méthodes sont aussi utilisées pour activer une nouvelle réplique d'un groupe d'objets muni de la réplication active. Par ailleurs, ces deux méthodes ne sont pas déclarées au niveau de l'interface IDL puisqu'elles ne sont jamais invoquées par des objets de l'application.

II.3.2. Aspect explicite et implicite dans la norme FT-CORBA

Afin d'identifier un groupe de répliques, la notion d'une référence de groupe (*Interoperable Object Group Reference, IOGR*) [OMG 2002b], dans la norme FT-CORBA, a été ajoutée à la notion d'une référence d'objet (*Interoperable Object Reference, IOR*), utilisée dans la norme CORBA standard,. En effet, un IOGR est un IOR à plusieurs profils auquel on ajoute des informations relatives aux changements d'appartenance au groupe, sa version et l'identificateur du primaire dans le cas d'une réplication passive. Les profils contenus dans l'IOGR renferment l'information nécessaire à l'ORB pour atteindre les différents objets CORBA du groupe de répliques. Lors de l'initialisation d'un groupe de répliques, seul l'identificateur du groupe (IOGR) est publié, les IOR des différentes répliques sont inaccessibles pour les clients. Ainsi, la norme FT-CORBA assure la transparence à la fois de la réplication des serveurs et du recouvrement des erreurs pour les clients. Malgré le fait que l'utilisation de l'IOGR est masquée au client, ce dernier invoque d'une manière explicite les différents membres d'un groupe de répliques. L'utilisation d'une référence de groupe facilite la

[11] *Push* où chaque réplique est amenée à envoyer un message au détecteur de fautes pour l'informer qu'elle n'est pas défaillante, et (2) *Pull* où le détecteur doit envoyer la requête `is_alive()`, méthode de l'interface *pullmonitorable* que chaque réplique doit hériter, pour savoir si les répliques sont encore en service.

diffusion des messages du client par rapport à l'utilisation d'un IOR, puisque les serveurs destinataires sont tous repérés par cet identificateur, et l'on n'est pas tenu de gérer plusieurs IOR pour pouvoir diffuser les messages aux différentes répliques. Néanmoins, cette technique explicite n'est pas applicable aux clients qui ne satisfont pas la norme FT-CORBA, appelés *clients légataires*, et où la référence de groupe n'est alors pas exploitable.

Pour résoudre le dernier objectif de la norme FT-CORBA, à savoir la permission aux clients légataires de bénéficier des plates-formes tolérants les fautes selon la norme FT-CORBA, l'utilisation d'une approche implicite est inévitable. En effet, lorsqu'un client demande la référence d'un objet (i.e. en utilisant le service de désignation ou un fichier renfermant les références) FT-CORBA lui fournit l'identificateur du groupe de réplique "IOGR" auquel le serveur appartient. Pour les clients légataires, cet IOR (i.e. IOGR) récupéré ne peut être utilisé pour faire parvenir ses messages au serveur cible. Ainsi, cet IOGR doit être utilisé d'une manière implicite pour que les messages issus des clients légataires puissent arriver aux différents membres d'un groupe de répliques. Une solution portable par rapport aux intergiciels a été proposée dans le cadre du projet IRL[12] (*Interoperable Replication Logic*) [Baldoni 2002, Baldoni 2003]. Cette solution consiste à munir les clients légataires et conformes à la norme CORBA 2.4 d'un intercepteur (*Outgoing Request GateWay Interceptor, ORGWInterceptor*) qui intercepte tout message sortant ou réponse au niveau du client. Le rôle de cet intercepteur est de voir si la référence que le client utilise est un IOR ou bien un IOGR, auquel cas, il relève une exception de re-direction de requête à un membre du groupe de réplique.

II.3.3. Aspect réflexif de la norme FT-CORBA

La norme FT-CORBA propose deux modes différents pour la gestion de la tolérance aux fautes. Dans le premier mode, la création d'un groupe de réplique, la gestion de ses membres, etc. est léguée à l'application. Tandis que dans le second mode, ces fonctionnalités sont à la charge de la plate-forme ce qui assure la séparation entre l'aspect fonctionnel de l'application (i.e. le service que propose l'application) et l'aspect non fonctionnel de la plate-forme (i.e. tolérance aux fautes).

[12] IRL est un projet en cours de réalisation au sein de l'université de Rome. Il a pour objectif d'étudier l'impact des architectures trois-tiers par rapport à la réplication des applications en implémentant la norme FT-CORBA d'une manière inter-opérationnelle et portable.

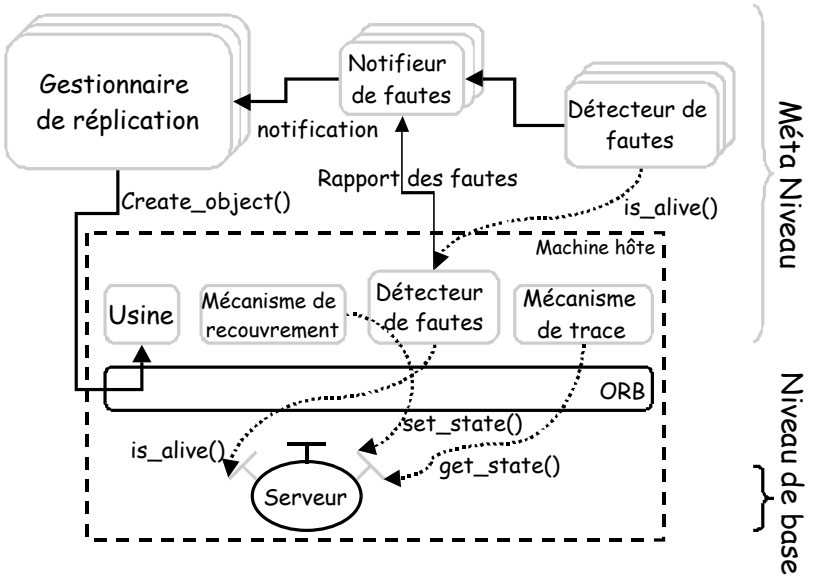


Figure 26. Réflexivité dans FT-CORBA

La Figure 26 présente une partie d'un groupe de répliques qui suit la norme FT-CORBA en mettant en avant les parties qui assurent la réflexivité. En effet, chaque serveur d'application (i.e. niveau de base) présente une interface fonctionnelle et une méta-interface composées des interfaces standards définies par la norme FT-CORBA `checkpointable`, `pullmonitorable` et `updateable`. Ces interfaces permettent au méta niveau de récupérer l'état du serveur avec `get_state()` / `get_update()`, de savoir s'il est toujours en service avec `is_alive()` (i.e. introspection), et de mettre à jour l'état du serveur avec `set_state()` / `set_update()` (i.e. intercession). Le mécanisme de trace ainsi que le détecteur de fautes sur la machine hôte, qui font partie du méta niveau, assurent la réification de l'état du serveur au gestionnaire de réplication.

II.4. CONCLUSION

Nous avons présenté dans ce chapitre les deux familles d'approches pour la tolérance aux fautes dans les systèmes informatiques répartis à base d'intergiciels et qui ont été à l'origine de la proposition de la norme FT-CORBA. La différence majeure entre les approches classiques et les approches réflexives est le souci de séparer l'aspect fonctionnel (i.e. le service rendu par l'application) de l'aspect non fonctionnel (i.e. les mécanismes de tolérance aux fautes).

Dans un premier temps, nous avons présenté les travaux qui s'inscrivent dans la famille des approches classiques. Nous avons affiné la classification proposée par Felber [Felber 1996], qui est basée sur la localisation du système de communication de groupe par rapport à l'ORB (i.e. intégration, service et interception). Notre apport réside dans l'ajout d'un critère qui correspond à l'implication du client de l'application dans le protocole de tolérance aux fautes. Nous avons fait la différence entre les approches implicites où la tolérance aux fautes est

totalement transparente pour le client et celles explicites où le client choisit le groupe de réplique auquel il adresse son message. Cette nouvelle dimension nous permet de mettre l'accent sur l'implication de l'utilisateur d'un système dans l'activation des mécanismes de tolérance aux fautes.

Dans la deuxième partie, nous avons présenté les travaux qui s'inscrivent dans la famille des approches réflexives, où les mécanismes de tolérance aux fautes sont totalement disjoints de l'application. Nous avons dégagé quatre classes qui correspondent à l'emplacement des mécanismes de tolérance aux fautes et de l'application par rapport à la couche intermédiaire et la couche applicative d'un système informatique. Nous avons pu montrer la capacité de chaque classe à pouvoir assurer les différents modes de réplications. Par ailleurs, nous avons pu dégager une classe <A,I>, où les mécanismes fonctionnels se localisent au niveau de la couche applicative et les mécanismes non-fonctionnels se trouvent au niveau de la couche intermédiaire, qui s'adapte bien aux trois types de réplication.

Enfin, nous avons présenté la norme FT-CORBA, qui est l'aboutissement logique face à l'intérêt des communautés scientifiques et industrielles pour doter les applications CORBA de la tolérance aux fautes. Cette nouvelle norme implique le développeur de l'application dans la mise en œuvre des mécanismes de tolérance aux fautes. Cependant, l'aspect réflexif tel qu'il apparaît dans la norme CORBA au travers des intercepteurs permet d'envisager le développement des systèmes tolérant les fautes en découplant l'aspect fonctionnel de celui non-fonctionnel. C'est cette approche que nous analysons dans le troisième chapitre.

CHAPITRE III

MECANISMES REFLEXIFS DANS CORBA ET APPLICATION A LA TOLERANCE AUX FAUTES

Dans les implémentations de la norme CORBA ou dans d'autres intergiciels telle que la machine virtuelle JAVA, on peut trouver deux types de mécanismes réflexifs: des mécanismes « standards » et des mécanismes « spécifiques ». Les mécanismes standards sont spécifiés par la norme et par conséquent on peut les trouver dans différentes implémentations d'intergiciels. Les mécanismes spécifiques sont propres à certaines implémentations ; ils ne sont pas supportés par l'ensemble des intergiciels car ils ne sont pas spécifiés au niveau de la norme. L'ensemble des travaux de recherche que nous avons présenté dans le chapitre précédent ont utilisé des mécanismes réflexifs spécifiques. Dans ce chapitre, nous évaluerons jusqu'où les mécanismes standards, présents dans la norme CORBA, peuvent être utilisés pour la tolérance aux fautes, en particulier pour la réplication des serveurs d'applications.

La première partie de ce chapitre est consacrée à la description des différents mécanismes réflexifs, spécifiques et standards au niveau des intergiciels. Nous mettrons en particulier l'accent sur les mécanismes standards qui seront utilisés pour le développement de la plate-forme DAISY. L'illustration des différents composants de cette plate-forme, destinée à munir des applications distribuées CORBA de mécanismes de réplication (i.e. passive, semi-active et active), fera l'objet de la seconde partie. Dans la troisième partie, nous présenterons deux approches différentes pour la mise en œuvre de la plate-forme DAISY. Ces deux approches se distinguent par l'emplacement des composants de tolérance aux fautes par rapport au support d'exécution, à savoir l'intergiciel. Nous terminons cette partie par la comparaison de ces deux approches sous l'angle de leur adaptabilité par rapport aux applications et de la complexité de leur mise en œuvre. Dans la quatrième partie, nous présenterons les limites conceptuelles, en termes architecturaux, des intercepteurs CORBA. Nous consacrerons la dernière partie de ce chapitre à la présentation de nos recommandations pour la définition d'une nouvelle génération d'intercepteurs compatibles avec les intercepteurs actuels. Ces nouveaux intercepteurs sont plus adaptés au développement des mécanismes non-fonctionnels, en particulier, ceux qui sont relatifs à la tolérance aux fautes.

III.1. MECANISMES REFLEXIFS DANS CORBA

L'intérêt croissant que suscite la réflexivité a influencé, dans un premier temps, quelques industriels qui ont doté leur intergiciels de mécanismes réflexifs spécifiques (i.e. propriétaires), et dans un second temps, l'OMG afin de munir la norme CORBA de mécanismes réflexifs standards. Nous présentons dans cette section tout d'abord les différents mécanismes spécifiques et ensuite les mécanismes standards.

III.1.1. Mécanismes réflexifs spécifiques

Servant de base pour le développement de l'intergiciel *DynamicTAO* (voir Section 2 du Chapitre II), *TAO* renferme la notion de mandataire intelligent *smart proxy* qui permet, comme nous allons le voir, le changement du comportement des applications. Quant à l'intergiciel ORBIX, qui a fait partie des premières approches implicites *ORBIX+ISIS* (voir Section 1 du Chapitre II), il propose d'une part la notion de filtres qui permettent le changement du comportement des applications, et d'autre part, les mécanismes de changement de requêtes et d'objets à l'exécution, ce qui permet de contrôler l'état des applications.

Dans une première partie, nous allons présenter les mécanismes affectant le comportement des applications, à savoir les filtres d'ORBIX et les mandataires intelligents de DynamicTAO. Ensuite, nous présenterons les transformateurs de requêtes et les chargeurs d'objets d'ORBIX qui permettent la modification de l'état des applications.

III.1.1.1. Notion de filtres

Il existe deux types de filtres définis par ORBIX : ceux qui sont associés aux requêtes et ceux qui sont associés aux objets [IONA 2000a]. Les premiers permettent d'ajouter des pré et/ou post-traitements lors de l'envoi des requêtes et à la réception des réponses. Ces filtres permettent d'attacher des données non fonctionnelles lors de l'émission d'une requête et de les récupérer lorsque la requête arrive à destination. Les filtres associés aux objets permettent de limiter les pré et/ou post- traitements aux requêtes envoyées à un serveur particulier. Ces filtres permettent d'observer le contenu des requêtes et des réponses (i.e. les paramètres).

En termes réflexifs, l'observation des requêtes qui transitent entre le client et le serveur relève de la réification. L'ajout de données à ces requêtes permet aussi de changer le comportement des serveurs cibles. En effet, l'analyse de ces données peut être à l'origine de la non-transmission de la requête au serveur, ce qui relève de l'intercession.

Malgré leur lien avec la notion de réflexivité, les filtres associés aux requêtes ne peuvent pas observer les données relatives aux messages qui transitent entre le client et le serveur. Ils peuvent observer l'interaction client-serveur, mais n'ont pas la possibilité d'analyser le contenu des requêtes. En plus, ils ne sont activés que si les deux entités communicantes ne se trouvent pas sur la même machine, puisqu'ils ne sont appelés que lors du départ ou de l'arrivée d'une invocation à un espace d'adressage particulier. Par ailleurs, les filtres associés aux objets, ne

peuvent pas modifier les données relatives aux requêtes ce qui limite leur potentiel en termes d'intercession.

III.1.1.2. Mandataires intelligents *Smart proxies*

Les mandataires intelligents ont été introduits par l'intergiciel TAO. Ce sont des mandataires (méta-objets) qui sont créés par une usine à objets, et qui remplacent d'une manière transparente les mandataires créés par défaut par le compilateur IDL [Wang 2001]. La réécriture des mandataires est matérialisée par le changement de leur comportement ; elle permet à un développeur d'application d'ajouter des paramètres aux requêtes, de tracer les messages échangés, sans changer le code du client ni celui du serveur.

Comme l'insertion du code non fonctionnel est assurée par la réécriture du mandataire, le développeur de tels mécanismes peut accéder au contenu des requêtes. Ainsi, tout mandataire intelligent est capable d'observer et de contrôler le contenu des messages qu'il intercepte, et par conséquent, d'observer et de changer le comportement des serveurs d'application.

Les mandataires intelligents sont spécifiques aux interfaces auxquelles ils sont associés. Ceci rend leur ré-utilisation limitée puisqu'il faut écrire un mandataire par service. De plus, l'utilisation de l'invocation dynamique des interfaces passe outre l'activation des mandataires intelligents ce qui les rend inopérants.

III.1.1.3. Transformateurs de requêtes

Les transformateurs de requêtes sont définis par l'intergiciel ORBIX. Ils fournissent une interface permettant la modification du tampon des données relatives à une requête `CORBA::Request` juste avant que l'invocation d'une opération ne soit activée au niveau du serveur, et juste avant que la réponse ne soit retournée au client. Cette modification peut être appliquée à toutes les requêtes sortantes d'un client particulier ou aux requêtes destinées à un serveur particulier [IONA 2000c].

Cette modification des requêtes est décomposée en deux mécanismes élémentaires : le premier est celui de l'observation des données qui seront empaquetées en une requête (i.e. réification), et le second est celui de la modification des données qui permet d'agir sur le comportement et l'état du serveur (i.e. intercession).

Ces transformateurs ne sont associés qu'aux objets émetteurs de messages (client) ; ainsi toute transformation du côté du récepteur (serveur) n'est pas permise. Par ailleurs, ces transformateurs ne sont pas activés pour des objets qui se trouvent sur la même machine. En effet, dans le cas des invocations locales, les requêtes sont directement transmises après empaquetage au serveur destinataire. Les transformations s'appliquant à des requêtes de type IIOP n'ont donc pas accès à ce type de requêtes, ce qui limite leur potentiel pour implémenter des mécanismes non-fonctionnels répartis.

III.1.1.4. Chargeurs d'objets à l'exécution

Les chargeurs d'objets sont définis par l'intergiciel ORBIX. Ils permettent de charger dynamiquement un objet CORBA à partir d'un contexte préalablement sauvegardé [IONA 2000b]. Lorsqu'un client invoque un serveur défaillant, le chargeur d'objet instantie un nouveau serveur qui se charge de retourner une réponse et non pas une exception au client. Le chargeur d'objet permet ainsi de masquer la défaillance des serveurs aux clients.

Le chargement des objets lors de l'exécution fait intervenir trois mécanismes réflexifs : la réification, l'intercession et l'introspection. Le premier mécanisme permet de rediriger les requêtes au chargeur de l'objet lorsque le serveur destinataire est défaillant. Le chargeur d'objets assure, ensuite, l'exécution du second mécanisme (i.e. intercession**)** qui se charge de la mise à jour de la référence et de l'état d'un nouveau serveur. Enfin, le chargeur d'objet utilise l'introspection pour obtenir l'état du serveur auquel il est associé (i.e. acquisition de points de reprise) et l'enregistre sur un support stable.

L'utilisation des chargeurs d'objets pose deux problèmes : l'intrusivité et l'implication du développeur de l'application. Le premier problème résulte de la déclaration des chargeurs d'objets au niveau des sources de l'application. Le second problème réside dans le fait que la capture de l'état est à la charge du développeur de l'application ce qui l'implique dans la mise en œuvre du mécanisme d'introspection et ceci ne relève pas forcément de ses compétences.

III.1.2. Mécanismes réflexifs standards

Lorsque l'objectif est de fournir une plate-forme réflexive utilisable pour toutes les applications distribuées basées sur un intergiciel conforme à la norme CORBA, il est nécessaire d'utiliser des mécanismes standards et non des mécanismes spécifiques à une implémentation particulière de la norme; l'utilisation de ces derniers limite l'interopérabilité. Nous nous sommes focalisés sur l'identification des mécanismes réflexifs offerts par ce type d'intergiciels. Nous proposons dans cette section une analyse de leur potentiel afin de les utiliser pour mettre en œuvre la réplication des services.

III.1.2.1. Comportement

L'OMG a défini deux types d'intercepteurs portables, l'intercepteur de références des objets CORBA (*IOR interceptor*) et l'intercepteur des requêtes (*request interceptor*).

L'intercepteur de références est activé lors de la création de la référence d'un objet CORBA pour y ajouter l'information nécessaire à l'utilisation correcte d'un service particulier dans un contexte de communication donné. Par exemple, si au lieu d'utiliser le protocole GIOP nous utilisons le protocole DCE-CIOP *Data Communication Equipement-Common InterORB Protocol*, les informations relatives à la référence DCE-CIOP d'un objet doivent être incluses dans l'IOR comme des données d'un nouveau profil d'utilisation (voir section 1 du chapitre I). Ces fonctionnalités ne permettent pas d'implémenter des mécanismes de tolérance aux fautes.

L'intercepteur des requêtes permet d'insérer un traitement particulier avant l'envoi et la réception des requêtes et des réponses, de rediriger les requêtes destinées à un premier serveur vers un second, ou de ne pas les délivrer.

C'est ce second type d'intercepteurs que nous mettrons en avant dans ce mémoire, puisque nous l'utiliserons afin d'ajouter les traitements nécessaires aux mécanismes de réplication d'une manière transparente. Dans un second temps, nous évaluerons leur potentiel en termes de réflexivité et vis-à-vis de leur utilisation dans le cadre de plates-formes tolérant les fautes.

Comme le montre la Figure 27, ce type d'intercepteur peut être utilisé pour traiter les requêtes à la fois du côté du client, et du côté du serveur. Nous nommons ceux qui se trouvent du côté client PIC *Client Portable Request Interceptors* et ceux qui se trouvent du côté serveur PIS *Server Portable Request Interceptors*.

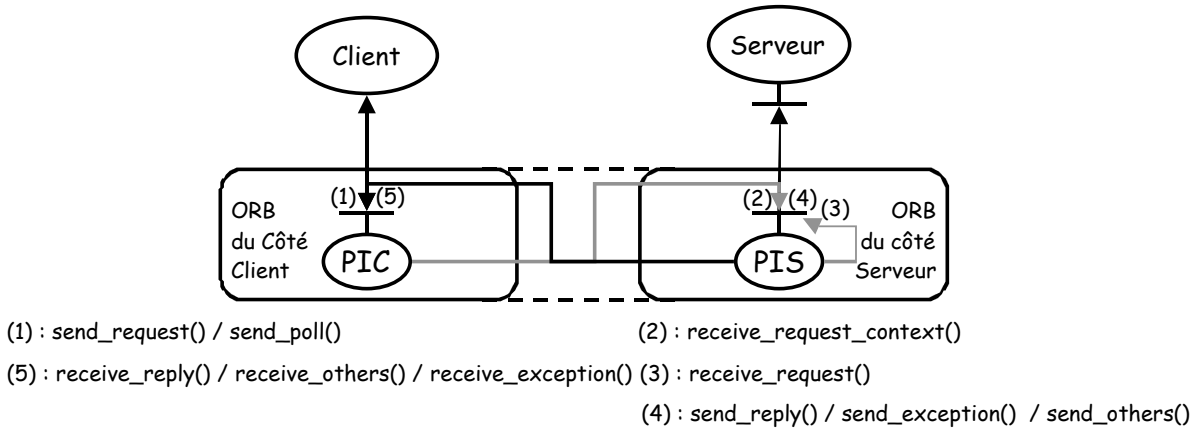


Figure 27. Interfaces des intercepteurs de requêtes *PIC* et *PIS*

Ce type d'intercepteur possède une interface fixe ; les différentes méthodes qui y sont définies représentent l'insertion des traitements à réaliser avant envoi et réception des requêtes et des réponses. Par ailleurs, l'activation de ces méthodes est assurée d'une manière automatique par l'ORB.

- Lorsque le client envoie une requête et que celle-ci est empaquetée dans un message GIOP, l'ORB appelle les méthodes `send_request` ou `send_poll` selon que la méthode est synchrone ou asynchrone (1).

- Une fois envoyée via le réseau, la requête arrive au niveau de l'ORB du côté serveur. Avant que la requête ne soit dépaquetée, l'ORB active la méthode `receive_request_context` (2). Lors de l'activation de cette méthode, les paramètres de la requête ne sont pas observables. En revanche, l'avantage de ce point d'interception est qu'il peut être exécuté dans un brin d'exécution (thread) différent de celui où la requête est exécutée.

- Avant que la requête n'arrive au niveau du squelette du serveur cible, l'ORB active la méthode `receive_request` du PIS (3). À ce niveau, les paramètres de la requête sont observables (mais pas modifiables).

- Au retour de la réponse, et lorsqu'elle est transformée en un message GIOP, l'ORB active les méthodes `send_reply` et `send_others` selon que le service réalisé est synchrone ou asynchrone (4). L'ORB active la méthode `send_exception` lorsque le serveur génère une exception durant l'élaboration d'une réponse. Cette méthode est également activée dans le cas où un problème de communication surgit lors du retour de la réponse et avant son interception par le PIS, ou même si une des quatre méthodes antérieures de l'interface du PIS déclenche une exception.

- Une fois que la réponse arrive au site client, et après que le message IIOP soit transformé en un message sous le format GIOP, l'ORB active les méthodes `receive_reply` ou `receive_others` si la réponse est bien arrivée, ou bien la méthode `receive_exception` si une exception a été soulevée pendant l'élaboration d'une réponse ou lors de la transmission du message (5).

Une fois activés, ces intercepteurs peuvent observer le contenu des messages qui transitent entre clients et serveurs (i.e. réification).

En outre, la norme CORBA a défini des méta-données standards qui peuvent être interprétées par les intercepteurs. Par exemple, du côté client, l'intercepteur PIC peut accéder aux données contenues dans la structure `ClientRequestInfo` et, du côté serveur, l'intercepteur PIS peut accéder aux données contenues dan la structure `ServerRequestInfo` [OMG 2002c]. Ces méta-données contiennent des informations relatives au nom du serveur cible, à la politique de transmission de la requête, à la liste des paramètres, etc.

Par ailleurs, ils peuvent rediriger une requête vers un serveur différent du serveur initial et ils peuvent inhiber l'invocation d'une requête (i.e. intercession comportementale). Par conséquent, les intercepteurs de requêtes CORBA peuvent être considérés pour l'instant comme une forme élémentaire de protocole à méta-objets permettant d'introduire des traitements non-fonctionnels. Par contre, ces intercepteurs ne peuvent pas observer directement l'objet auquel ils sont associés (i.e. ils ne fournissent pas le mécanisme d'introspection) et ne peuvent pas agir directement sur l'état d'un objet (i.e. ils ne fournissent pas le mécanisme d'intercession structurelle) comme le résume le Tableau 6.

Mécanismes Réflexifs	Réification	Intercession Comportementale	Intercession Structurelle	Introspection
Disponibilité	Oui	Oui	Non	Non

Tableau 6. Mécanismes réflexifs offerts par les intercepteurs CORBA

III.1.2.2. État

La communication entre les deux niveaux d'une architecture réflexive (i.e. base et méta) est réalisée à travers leurs méta-interfaces. C'est en particulier le cas pour le gestion de l'état des objets. Les objets doivent fournir une méta-interface permettant de capturer ou de restaurer leur état. L'héritage est un moyen permettant de munir ces applications d'une telle méta-interface de manière «transparente» du point de vue des développeurs d'applications.

Laisser l'implémentation de ces méthodes aux soins du développeur de l'application semble la solution la plus simple, d'autant que c'est lui qui connaît le mieux l'organisation de l'application. Cependant, impliquer le développeur de l'application dans la mise en œuvre des mécanismes de tolérance aux fautes est une source potentielle d'erreur. De plus, une telle solution risque de ne pas être réutilisable pour différentes applications, dans le sens où elle est dépendante de l'application.

Une solution générique et s'appuyant sur des outils automatiques est préférable. L'utilisation de la sérialisation JAVA permet en effet de récupérer et de mettre à jour l'état d'une application d'une manière standard sans faire appel à l'intervention de son développeur.

La Figure 28 montre comment sérialiser l'état d'un objet serveur.

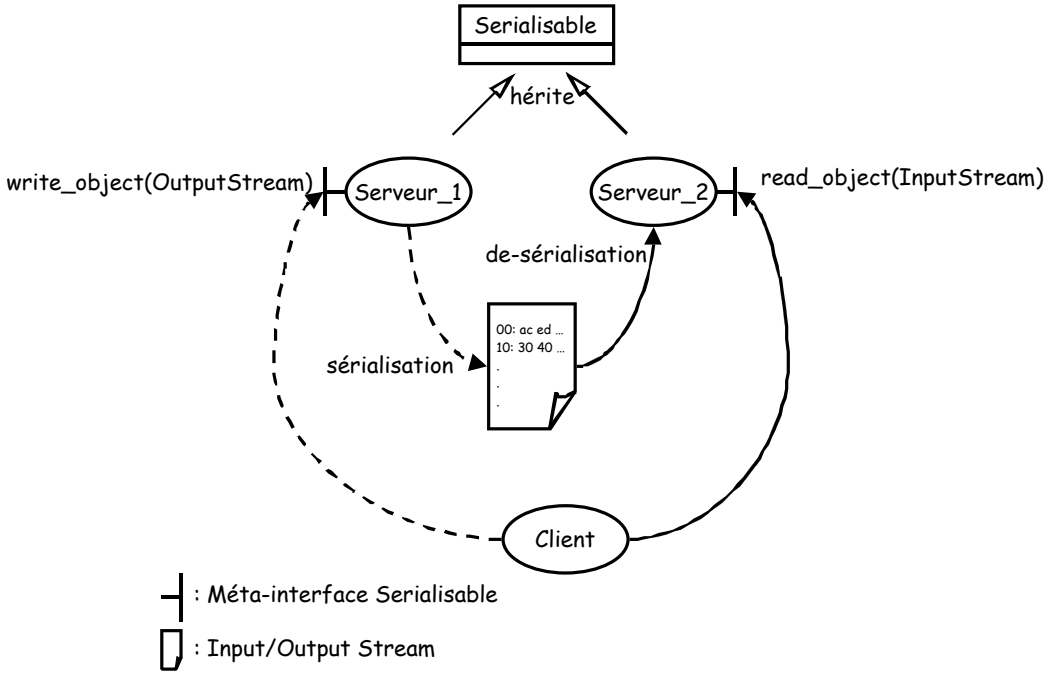


Figure 28. Sérialisation et dé-sérialisation des objets

Pour sérialiser l'état du *Serveur_1*, ce dernier doit hériter de l'interface *Serialisable* offerte par Java. La machine virtuelle Java transforme les informations relatives à l'état de l'objet en une séquence d'octets en utilisant la méthode `write_object()` et l'enregistre sur un support stable *Input/Output Stream*. Inversement, cette séquence peut être utilisée pour initialiser un objet ou mettre à jour un objet déjà existant *Serveur_2*, en dé-sérialisant les informations extraites du support stable en utilisant la méthode `read_object()`. L'aspect bijectif et

standard de la transformation de l'état un objet est assuré, d'une manière formelle, en s'appuyant sur une grammaire [Sun 1998].

Comme nous allons le montrer, nous utilisons cette technique pour la sérialisation et la désérialisation de l'état des serveurs. Ceci permet d'avoir, d'une part, une certaine confiance en la récupération et la mise à jour de l'état, et d'autre part, une manière standard et indépendante de l'application pour la gestion de l'état.

III.2. ARCHITECTURE DE DAISY

DAISY *Dependable Adaptative Interceptors and Serialization-based sYstem* est une plate-forme fournissant des mécanismes de tolérance aux fautes distribués sur le bus à objets CORBA. Nous allons présenter dans cette section l'architecture de cette plate-forme que nous avons conçue dans le but d'analyser les limites des mécanismes réflexifs standards.

III.2.1. Spécifications

La plate-forme DAISY fournit les mécanismes de réplications classiques (cf. Figure 29) : la réplication passive, la réplication semi-active et la réplication active, pour étendre le comportement des applications client/serveur. L'administrateur initialise l'application serveur en choisissant un mode de réplication parmi les trois offerts par la plate-forme DAISY. Le client, pour sa part, initialise l'application cliente puis demande à travers cette application les services offerts par l'application serveur.

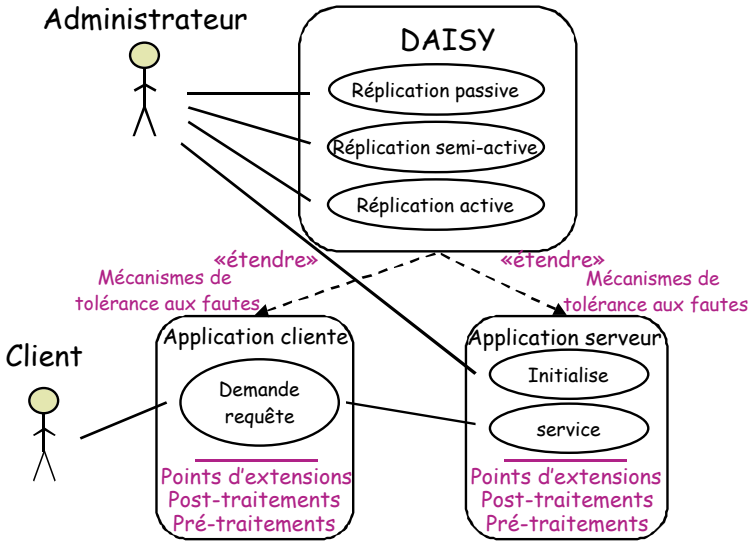


Figure 29. Diagramme d'utilisation

Comme nous l'avons vu dans le premier chapitre, les réplications passive et semi-active répondent aux mêmes modèles de fautes. En effet, ces deux mécanismes ne tolèrent que les défaillances par arrêt d'un serveur.

La réplication active possède un modèle de faute différent des autres types de réplication. En effet, elle peut tolérer aussi bien les défaillances par arrêt que les défaillances par valeur. La tolérance des fautes en valeur nécessite un vote sur les sorties des répliques ; cela implique que ces serveurs doivent être déterministes. Afin d'assurer ce déterminisme, la plate-forme doit garantir la diffusion atomique des requêtes et la gestion de l'appartenance d'un groupe de répliques.

La plate-forme DAISY est indépendante et transparente. En effet, nous n'avons pas besoin d'ajuster notre plate-forme puisque sa mise en œuvre repose sur des composants et des mécanismes standards. Par ailleurs, l'utilisation de cette plate-forme est transparente par rapport aux serveurs d'applications qui seront répliqués et dont l'architecture ne sera pas altérée pour y inclure les mécanismes de réplication *separation of conserns*. De même, cette plate-forme est transparente par rapport aux clients qui ne s'aperçoivent ni du traitement sûr (par plusieurs serveurs d'application) de leurs requêtes émises ni de la défaillance d'une réplique.

En outre, la mise en œuvre de cette plate-forme permettra d'évaluer l'utilisation des composants réflexifs sur étagères, en l'occurrence les intercepteurs CORBA et la sérialisation java, pour la mise en œuvre des mécanismes de réplication. Les avantages et les inconvénients de ces composants par rapport à leur utilisation dans les systèmes sûrs de fonctionnement seront discutés aussi bien sur le plan architectural que sur le plan de l'implémentation.

III.2.2. Définitions (principes)

La Figure 30 montre l'architecture à composants de DAISY.

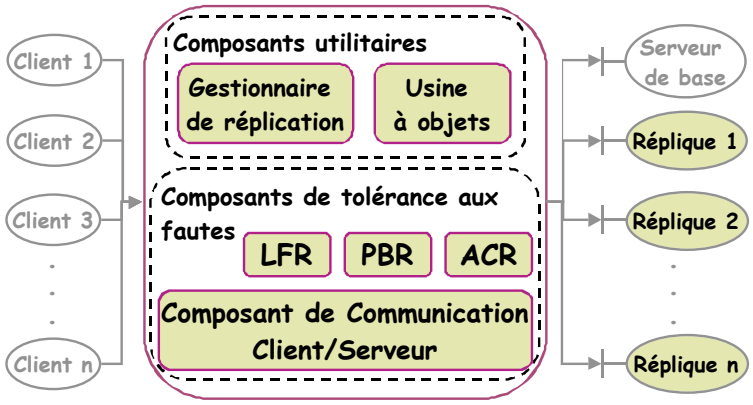


Figure 30. Vue Générale de l'architecture

L'ensemble de ces composants est formé de deux familles différentes : la première famille correspond aux composants de tolérance aux fautes et la seconde aux composants utilitaires. La conjonction de ces deux familles de composants a pour objectif de faire traiter toute requête, émise par un client, par un ou plusieurs serveurs répliqués afin d'aboutir aux contraintes de tolérance aux fautes dictées par l'administrateur de la plate-forme.

En effet, la première famille de composants assure la mise en œuvre des mécanismes de réplication. Cette mise en œuvre est assurée par l'action des composants de cette famille sur les messages qui circulent entre les clients et les serveurs, d'une part, et le comportement des serveurs d'application d'autre part. Les composants utilitaires, qui font partie de la seconde famille, assurent le contrôle de la plate-forme, en particulier le déploiement des composants de tolérance aux fautes, et le contrôle des serveurs d'application qui font appel à notre plate-forme.

L'utilisation d'une telle architecture (i.e. par composants) permet de séparer d'une manière claire les différents types de mécanismes. L'activation d'un mode de réplication n'entraîne pas l'activation des composants assurant les deux autres modes. Ainsi, si une erreur s'est introduite dans le code d'un type de réplication, elle n'aura pas d'effet sur le code des deux autres. Ceci limite la propagation des erreurs entre les différents composants. Par ailleurs, cette décomposition se prête mieux à l'utilisation de composants génériques qui sont indépendants des applications auxquelles ils sont associés (i.e. changement de l'application qui est associée au serveur de base et les applications clientes qui l'utilisent).

III.2.3. Composants de tolérance aux fautes

Nous distinguons quatre composants de tolérance aux fautes qui sont : le composant de la communication client/serveur, le composant LFR *Leader Follwer Replication*, le composant PBR *Primary Backup Replication* et celui ACR *ACtive Replication*. Le premier composant assure l'aspect relatif à la communication entre le client et le serveur. Les mécanismes relatifs à la réplication semi-active sont, quant à eux, agrégés au niveau du composant LFR. Le composant PBR est responsable de la mise en œuvre des mécanismes de la réplication passive. Le quatrième et dernier composant, ACR assure la réplication active.

Nous allons utiliser la réflexivité pour mettre en œuvre les mécanismes non fonctionnels, nous tenons à préciser que les composants de tolérance aux fautes formeront les méta-objets des différentes applications auxquelles ils fourniront les mécanismes de réplication classiques.

III.2.3.1. Composant de communication client/serveur

Le composant de communication C/S est propre à chaque client et a pour charge d'assurer la transparence de l'envoi des requêtes par rapport au client, l'identification de chaque requête sortante, la récupération des vues du groupe de répliques et la protection du client envers les défaillances des serveurs.

Puisque le CC-C/S assure le rôle d'un mandataire pour les mécanismes de tolérance aux fautes, fournies par les composants ACR, PBR et LFR, il rend ces mécanismes transparents par rapport au client. Dans le cas d'un fonctionnement en absence de fautes, avant qu'une requête ne soit délivrée, elle est identifiée d'une manière unique pour faciliter son repérage par les autres composants de tolérance aux fautes. Dans le cas où un des serveurs membres du

groupe de réplique défaillerait par arrêt, le CC-C/S masque cette défaillance en essayant d'établir un nouveau lien de communication avec un autre serveur qui fonctionne.

Comme notre objectif est de voir les limites des composants réflexifs standards, nous n'avons pas axé notre effort sur la fourniture ou l'utilisation de protocoles de groupes. Ainsi, le composant de communication client/serveur n'assure pas les fonctionnalités d'un système de communication de groupe *GCS* ; par exemple, ce composant n'assure ni la communication atomique ni la diffusion des messages.

Par ailleurs, l'envoi des requêtes aux différentes répliques est réalisé par la plate-forme DAISY, au niveau des composants de tolérance aux fautes et sans intervention du client, nous pouvons donc inscrire cette architecture parmi les approches dites implicites (voir la Section 1 du Chapitre II).

III.2.3.2. Composants de réplication passive et semi-active

Les composants PBR et LFR répondent, d'une part, touts les deux au même modèle de fautes à tolérer et, d'autres part, possèdent chacun un ensemble de répliques qui se décline en deux types de serveurs. En effet, la réplication passive et la réplication semi-active ne tolèrent que les défaillances par arrêt des serveurs. Les deux types de serveurs sont : le serveur primaire et les serveurs répliques dans le cas de la réplication passive, et le serveur meneur et les serveurs suiveurs dans le cas de la réplication semi-active. Par ailleurs, la seule différence qui existe entre les deux composants est que le premier envoie un état, tandis que le second envoie des messages de synchronisation.

Dans le cas de la réplication passive, la requête envoyée par le composant de communication arrive au niveau du composant PBR associé au serveur primaire. Le composant PBR délivre la requête au serveur d'application, et au retour de la réponse, il récupère l'état du serveur primaire pour l'envoyer aux serveurs répliques pour qu'ils mettent à jour leurs états. Si le composant PBR est associé à un serveur réplique, il est alors chargé d'une part de la mise à jour de l'état du serveur répliqué, et d'autre part, lors de l'occurrence d'une défaillance par arrêt du primaire, il assure la transition vers le nouveau primaire.

Dans le cas de la réplication semi-active, la requête diffusée par le composant de communication arrive au niveau des différents composants LFR. Ces composants sont associés d'une part au serveur meneur, et d'autre part, aux serveurs suiveurs. Dans le premier cas, le composant LFR envoie des messages de synchronisation pour imposer un ordre identique pour le traitement des requêtes au niveau des différents suiveurs. Dans le cas où le composant LFR se trouve du côté d'un serveur suiveur, il met en attente toutes les requêtes jusqu'à ce qu'il reçoive l'ordre de synchronisation. Par ailleurs, ce composant agit de la même façon que le composant PBR du côté réplique, c'est-à-dire, il change de comportement lorsque le serveur auquel il est associé se trouve élu comme nouveau meneur en cas de défaillance.

III.2.3.3. Composants de réplication active

Les composants ACR implémentent un protocole de réplication active. Ils assurent la diffusion des messages en respectant un ordre total. De plus, comme la réplication active tolère les défaillances par valeurs, ces composants assurent le vote pour pallier à de tels problèmes.

Lorsque l'application évolue en absence de fautes, nous utilisons un protocole d'ordre asymétrique [Ezhilchelvan 1995] pour assurer un ordre total pour la délivrance des requêtes. Ce type de protocole utilise un membre du groupe, que nous appelons principal, pour séquencer et ordonner les messages. Ainsi, lorsque la requête arrive au niveau du composant ACR du serveur principal, il la diffuse aux différents ACR des serveurs répliques (subsidiaires). Ensuite, lors du retour des différentes réponses, le composant ACR du serveur principal assure le vote et retourne la réponse majoritaire au client.

Lorsqu'un serveur, principal ou subsidiaire, défaille en valeur, le composant ACR du serveur principal réinitialise son état à partir de l'état d'un autre serveur jugé correct.

Lorsqu'un serveur subsidiaire défaille, l'ACR du serveur principal se charge de sa détection puis notifie le gestionnaire de réplication de cette défaillance par arrêt. En revanche, si le serveur principal défaille, c'est le composant ACR d'un autre serveur subsidiaire qui le détecte, s'assure de cette défaillance et notifie par la suite le gestionnaire de réplication.

III.2.4. Composants utilitaires

Nous distinguons deux composants utilitaires qui sont : l'usine à objets et le gestionnaire de réplication. Le premier composant permet la création, d'une manière automatique, des serveurs d'application. Le second composant assure, quant à lui, la gestion de la plate-forme DAISY en termes de composants de tolérance aux fautes ainsi que la gestion de l'ensemble des répliques.

III.2.4.1. L'usine à objets

L'usine à objets permet d'automatiser la création de répliques. L'objectif de cette automatisation est de pallier la dégradation du niveau de tolérance aux fautes lors de l'occurrence d'une défaillance d'une réplique de l'application. L'arrêt d'un serveur peut être provoqué par l'application des mécanismes de tolérance aux fautes, c'est à dire commandé par un composant de tolérance aux fautes, que nous allons présenter dans le chapitre suivant, ou « naturel » suite à l'activation d'une faute latente.

En effet, dans le cas d'une réplication passive, et si lors de l'initialisation nous créons N répliques d'un service donné, notre système peut tolérer N-1 défaillances simultanées par arrêt. Par conséquent, si une réplique défaillante n'est pas remplacée, notre système ne tolérera plus que N-2 défaillances simultanées par arrêt. Similairement, dans le cas d'une réplication active, si lors de l'initialisation nous créons 2N+1 répliques d'un service donné, notre système

peut tolérer jusqu'à N défaillances par valeurs simultanément, dans le cas où tous les serveurs répliques retournent une réponse dans le délai prédéfini. Par conséquent, si une réplique arrêtée n'est pas remplacée, le système ne tolère plus que N-1 défaillances par valeurs simultanément.

Par ailleurs, l'utilisation de l'usine à objets permet de masquer à l'administrateur du système l'insertion de la plate-forme lors du lancement d'une application. Ceci protège l'ensemble de l'application d'une mauvaise initialisation qui peut compromettre la mise en œuvre des mécanismes de tolérance aux fautes.

III.2.4.2. Le gestionnaire de réplication

Le gestionnaire de réplication fournit une interface à l'administrateur pour pouvoir choisir le mode de réplication initial, les paramètres du modèle temporel du système, le nombre de répliques à déployer et les machines qui vont héberger les différentes répliques. En outre, le gestionnaire de réplication peut changer d'un mode de réplication à un autre selon les conditions de fonctionnement des différentes répliques et les attributs initiaux de la tolérance aux fautes.

En effet, les différents paramètres saisis par le gestionnaire de réplication influencent le comportement du système global, c'est-à-dire la plate-forme, les serveurs d'application et les clients. Au début, le choix du type de réplication fixe le mode de défaillance (par valeur et/ou par arrêt) toléré. Ensuite, le choix du modèle temporel indique les délais de détection d'une défaillance dans le modèle asynchrone temporisé. Par la suite, le choix du nombre de répliques indique le nombre de fautes simultanées que la plate-forme peut tolérer. Enfin, le choix des machines qui hébergeront les répliques différentes, donne un indice sur le degré de distribution (ou décentralisation) qui peut refléter d'une part la charge des différentes machines impliquées, et d'autre part, le degré de corrélation entre les défaillances des différentes répliques. En somme, si le degré de distribution est égal à 1, c'est-à-dire chaque machine n'héberge qu'un seul serveur, ceci charge « à priori » équitablement les différentes machines. Cette charge équitable peut protéger l'ensemble du système de mauvaises décisions en déclarant des serveurs lents par rapport aux délais imposés par le modèle temporel comme défaillant, et ce faute de ressources. Par ailleurs, dans le cas où deux répliques résideraient sur la même machine, la défaillance d'un serveur peut corrompre l'espace mémoire de la machine hôte et compromettre le bon fonctionnement de la deuxième réplique. En plus, la défaillance matérielle de la machine hôte altère les propriétés de tolérance aux fautes du système entier, en termes de nombre de défaillances simultanées tolérées, plus que dans le cas où chaque machine héberge une seule réplique.

Dans la prochaine section, nous allons spécialiser l'architecture à composants définie précédemment en utilisant des composants génériques, à savoir les intercepteurs CORBA. Un intérêt particulier sera apporté à la mise en œuvre des composants de tolérance aux fautes. Nous allons présenter deux variantes de l'architecture DAISY qui varient selon la localisation des composants de la tolérance aux fautes par rapport à l'ORB.

III.3. DIFFERENTES VARIANTES DE DAISY

Nous présentons dans cette partie deux approches différentes pour mettre en œuvre l'architecture de DAISY. Dans la première approche, les composants de tolérance aux fautes sont présentés sous forme de services, c'est-à-dire à l'extérieur de l'ORB. En revanche, dans la seconde approche, ces composants sont intégrés dans l'intergiciel. Nous situons par la suite ces deux approches par rapport aux travaux présentés dans le chapitre précédent. Enfin, nous comparons ces deux approches entre elles, par rapport à leur réutilisation et leur transparence afin de choisir celle que développons dans le chapitre suivant.

III.3.1. Approche à base de composants externes

Dans cette approche, nous disposons les composants de tolérance aux fautes à l'extérieur de l'ORB (voir la Figure 31).

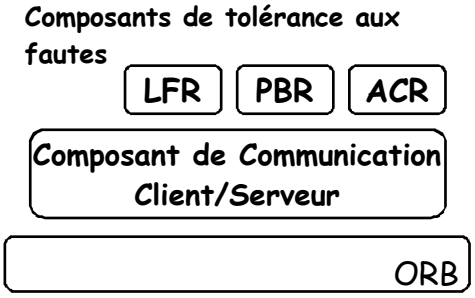


Figure 31. Composants de tolérance aux fautes externes à l'ORB

Comme nous utilisons la réflexivité pour mettre en œuvre la réplication des applications, ces composants seront alors vus comme des méta-objets. Nous avons considéré, dans cette première approche, que l'intercepteur de requêtes est à la base de la construction d'un protocole à méta-objet; par conséquent, il lie l'application aux composants de tolérance aux fautes et assure le transfert des informations et des requêtes entre ces deux niveaux.

III.3.1.1. Principe

La Figure 32 montre comment nous allons utiliser l'intercepteur de requêtes CORBA pour mettre en œuvre une architecture à composants externes à l'ORB.

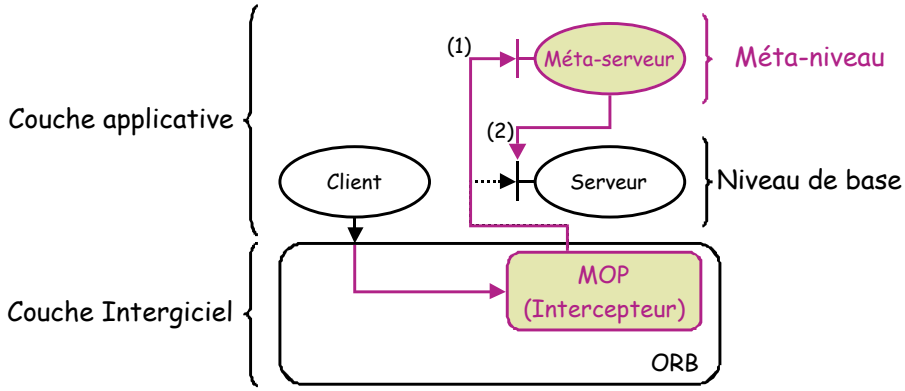


Figure 32. Protocole à méta-objets sur étagère (Intercepteur CORBA)

Dans cette première variante de DAISY, nous considérons l'intercepteur comme un support pour mettre en œuvre un protocole à méta-objets *MOP* à l'exécution [Killijian 2002]. Avec ce type de protocole, le lien entre serveur (i.e. Niveau de base) et le méta-serveur qui lui est associé (i.e. Méta-niveau) ainsi que les lois qui régissent la réification (1) et l'intercession (2) sont définies lors de l'exécution de l'application.

En effet, la création du lien entre le méta-serveur et le serveur est définie au niveau de l'intercepteur associé à ce dernier. Lors de l'initialisation de l'ORB, l'intercepteur de requêtes récupère l'IOR du méta-serveur pour initialiser le lien entre le niveau de base et le méta-niveau. Ce lien est, par la suite, activé chaque fois qu'une requête est interceptée.

Les lois qui régissent la réification et l'intercession sont aussi définies au niveau de l'intercepteur à l'exécution. Lors de l'interception d'une requête à destination du serveur, l'intercepteur peut choisir de la rediriger au méta-serveur en soulevant l'exception `ForwardRequest` ou de la transmettre au serveur destinataire.

En utilisant une telle approche, le méta-serveur peut aussi bien invoquer le serveur cible qu'élaborer une réponse et la retourner au client d'une manière transparente, c'est-à-dire sans que le client ni le serveur cible ne s'en aperçoivent.

La différence entre les protocoles à méta-objets à l'exécution, tels que CLOS [Kiczales 1991] ou OpenC++v1 [Chiba 1993], par rapport à un MOP à base d'intercepteurs réside dans leur niveau de définition au sein d'un système. Les premiers sont définis au niveau du langage, c'est-à-dire dans la couche applicative, en revanche, le second est définit au niveau de l'intergiciel.

III.3.1.2. Prototype de la réplication passive

Nous présentons dans la Figure 33 l'architecture de la variante à base de méta-objets externes de la plate-forme DAISY mettant en œuvre la réplication passive.

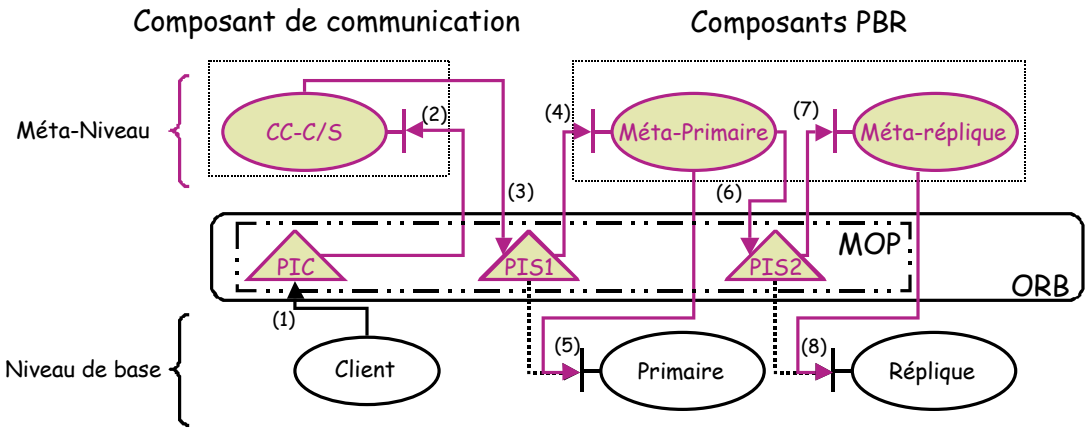


Figure 33. Réplication passive à méta-objets externes

Notre architecture est composée de deux niveaux : le méta niveau et le niveau de base. Le méta-niveau renferme les composants de tolérance aux fautes, à savoir les composants PBR et le composant de communication client/serveur *CC-C/S*. Ces composants, qui sont des méta-objets, sont aussi des objets CORBA identifiés d'une manière unique à travers un IOR. Le niveau de base est composé d'un client, d'un serveur primaire et d'un serveur réplique. En observant et en contrôlant le comportement et l'état des objets du niveau de base, les composants de tolérance aux fautes leur fournissent les mécanismes de la réplication passive.

Le lien entre le niveau de base et le méta-niveau est assuré par un protocole à méta-objet composé de trois intercepteurs : *PIC*, *PIS1* et *PIS2*.

Lors de l'initialisation des objets du niveau de base, l'intercepteur PIC récupère l'identificateur du composant de communication client/serveur pour le lier avec l'objet *client*. Ensuite, l'intercepteur PIS1 récupère l'identificateur du serveur *méta-primaire* pour le lier avec le serveur primaire. Enfin, l'intercepteur PIS2 assure l'association entre le méta-objet *méta-réplique* et l'objet *réplique*. L'ensemble de ces actions représente la phase d'initialisation du protocole à méta-objet (i.e. le lien entre le niveau de base et le méta-niveau).

Lors de l'exécution de l'application, toute requête issue du client est interceptée par le PIC (1) qui la réifie au CC-C/S (2). Ce dernier envoie ces requêtes au serveur primaire qui seront interceptées par le PIS1 (3). Celui-ci les réifie au serveur méta-primaire (4). Ce méta-objet traite les requêtes qui lui sont parvenues par l'intermédiaire du PIS1 puis les transmet au serveur *Primaire* (5). Par ailleurs, le méta-primaire récupère l'état du serveur primaire (6) puis élabore une requête de mise à jour de l'état du serveur réplique. L'intercepteur PIS2 réifie cette requête au serveur méta-réplique (9) qui la traite, puis assure la mise à jour du serveur réplique (10). Cette phase représente le mode de fonctionnement du protocole à méta-objet qui se résume :

i)- en la réification de toute requête sortante d'un client vers le méta-objet qui lui est associé.

ii)- en la réification de toute requête à destination d'un serveur vers son méta-objet associé.

iii)- en la transmission de toute requête à destination d'un objet et provenant de son méta-objet. Cette transmission concerne les requêtes simples (i.e. intercession comportementale) la récupération de l'état (i.e. introspection) et la mise à jour de l'état (i.e. intercession structurelle).

C1 : Problème et incidence sur la conception de DAISY

L'intercepteur ne permet pas au méta-primaire de récupérer ou de contrôler la structure de l'objet de base ni la définition de son interface. C'est pour cette raison qu'une application doit hériter les méthodes d'observations et de contrôles.

L'existence des intercepteurs CORBA est liée aux objets auxquels ils sont associés lors de l'initialisation (i.e. serveur et client); l'arrêt d'un serveur provoque la destruction immédiate du lien qui le lie à son méta-serveur. Il en est de même pour le lien entre le client et le serveur CC-C/S. Nous soulignons qu'un méta-objet dont l'objet est détruit peut être lié à un nouvel objet; Par ailleurs, chaque intercepteur est capable de détecter la défaillance d'un méta-objet.

III.3.1.3. Prototype de la réplication active

Nous présentons dans la Figure 34 l'architecture de la variante à base de méta-objets externes de la plate-forme DAISY, dans laquelle nous traitons la mise en œuvre de la réplication active.

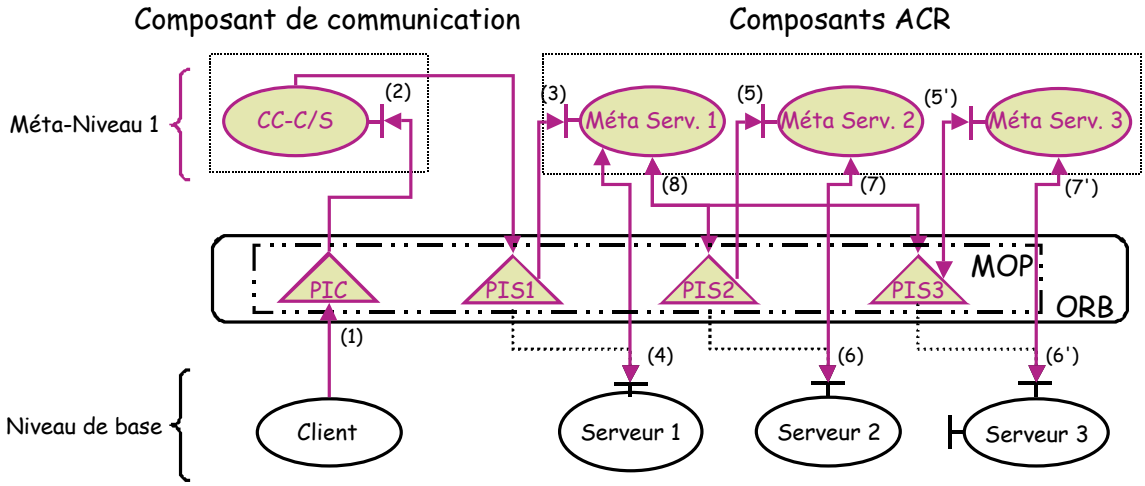


Figure 34. Réplication active à méta-objets externes

Comme pour la réplication passive, le méta-niveau renferme les composants de tolérance aux fautes, à savoir les composants ACR et le composant de communication client/serveur. Le niveau de base est composé d'un client et de trois serveurs d'applications : le *Serveur 1*, le *Serveur 2* et le *Serveur 3*. Ces trois serveurs forment le groupe de répliques.

Le lien entre le niveau de base et le méta-niveau est assuré par un protocole à méta-objet composé de quatre intercepteurs : *PIC*, *PIS1*, *PIS2* et *PIS3*.

A l'initialisation, l'intercepteur PIC récupère l'identificateur du composant de communication client/serveur pour le lier avec l'objet client. Ensuite, chaque intercepteur assure l'association entre l'objet et le méta-objet qui l'observe et le contrôle, par exemple PIS1 lie l'objet Serveur 1 au méta-objet *Méta Serv.1*. Cette phase représente la phase d'initialisation du protocole à méta-objet.

Lors de l'exécution de l'application, toute requête issue du client est interceptée par le PIC (1) qui la réifie au *CC-C/S* (2). Celui-ci envoie ces requêtes au serveur Serveur 1 qui seront réifiées par le PIS1 au Méta Serv.1 (3). Ce méta-objet traite les requêtes qui lui sont parvenues puis les transmet à l'objet du niveau de base qui lui est associé Serveur 1 (4). De plus, le Méta Serv.1 diffuse aux autres serveurs du groupe de répliques toutes les requêtes qui lui parviennent. Ces requêtes sont réifiées par les intercepteurs PIS2 et PIS3 aux méta-objets Méta Serv.2 et Méta Serv.3 (5, 5'). Chaque méta-serveur invoque par la suite l'objet du niveau de base qui lui est associé (6, 6'). Après avoir traité les requêtes, les serveurs retournent leurs réponses au méta-objet Méta Serv.1 (8) en passant par leurs méta-objets respectifs (7, 7'). Enfin, le Méta Serv.1 réalise le vote et retourne la réponse majoritaire au client.

III.3.1.4. Classification

Les clients qui utilisent cette plate-forme adressent leurs requêtes à un serveur particulier. Ces requêtes seront par la suite prises en compte par le protocole à méta-objets qui les redirigent vers les composants de tolérances aux fautes. Ces derniers assurent le traitement des requêtes émises par les clients. Comme les clients n'adressent pas explicitement leurs requêtes aux méta-objets, nous classons la plate-forme à composants de tolérance aux fautes externes parmi les approches implicites. De plus, les méta-objets dans cette plate-forme sont des objets CORBA offrant chacun un service particulier ; la coopération de ces méta-objets assure la réplication passive ou active du service de base. Par conséquent, nous classons cette plate-forme parmi les approches de tolérance aux fautes par service implicite.

Par ailleurs, nous classons la plate-forme à composants externes parmi les approches de tolérance aux fautes réflexives de la famille <A,A>. En effet, les méta-objets qui forment le méta-niveau et qui fournissent les mécanismes de réplication font partie de la couche applicative tout comme les objets du niveau de base. Cette approche est dotée d'un potentiel d'observation et de contrôle à large spectre. Par exemple, un méta-serveur peut changer le contenu d'une requête.

III.3.2. Approche à méta-objets intégrés

Contrairement à la première approche, dans cette seconde variante, nous intégrons les composants de tolérance aux fautes à l'intergiciel. Plus précisément, c'est au niveau des intercepteurs de requêtes que les mécanismes définis dans la section 2 sont réalisés. Comme

ces intercepteurs fournissent des mécanismes non-fonctionnels, ils sont considérés, ici, comme des méta-objets chargés de fournir les mécanismes de réplication de services (voir Figure 35).

Figure 35. Disposition des composants de tolérance aux fautes dans la seconde approche

III.3.2.1. Principe

La Figure 36 illustre l'utilisation de l'intercepteur de requêtes CORBA pour mettre en œuvre une architecture à composants intégrés à l'ORB.

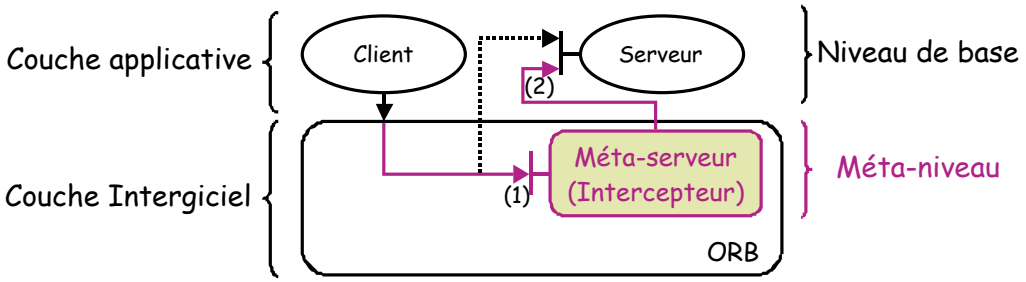


Figure 36. Méta-objet sur étagère (Intercepteur CORBA)

Dans cette seconde variante de DAISY, nous considérons l'intercepteur comme un support pour mettre en œuvre une stratégie de réplication de services (active ou passive).

En effet, toute requête émise par un client est réifiée par l'ORB à l'intercepteur de requêtes (1). Cette réification est assurée par l'activation de l'une des méthodes appartenant à son interface qui sont indépendantes de l'application. En d'autres termes, cette interface ne dépend pas des objets du niveau de base, ni de leurs interfaces ; elle n'est pas modifiable. L'intercepteur est un méta-objet particulier, dans le sens où, il ne peut pas modifier le contenu des requêtes qui lui sont réifiées ; son action se limite à l'observation de l'information qu'elles véhiculent. Les seules actions autorisées par la norme CORBA aux intercepteurs sont relatives à l'intercession comportementale (2). Cette dernière se matérialise en la transmission de toute requête réifiée au serveur cible, ou bien en sa re-direction vers un autre serveur équivalent, ce qui veut dire, vers un serveur qui possède la même interface que le serveur original [Bennani 2003].

C2 : Problème et incidence sur la conception de DAISY

L'intercepteur possède une méta-interface non modifiable. C'est pour cette raison que les méthodes destinées à l'extension de cette interface doivent être héritées par l'application.

Nous avons souligné dans la section III.1.2.1 qu'il existe un intercepteur de requêtes du côté client et un autre du côté serveur. Nous stipulons ainsi que tout intercepteur qui se trouve du côté client représente le méta-objet de la souche du client (i.e. méta-stub). En revanche, l'intercepteur du côté serveur est le méta-objet du serveur (i.e. méta-serveur).

C3 : Problème et incidence sur la conception de DAISY

L'intercepteur ne peut pas avoir de méta-objet (i.e. méta-intercepteur). C'est pour cette raison qu'on ne peut pas observer et changer son comportement.

Comme le lien entre objet et méta-objet doit être régi par un protocole *MOP*, la norme CORBA en a défini un à l'exécution.

En effet, l'établissement de lien entre objet et méta-objet (i.e. la sélection) se fait lors du chargement de l'application. L'intergiciel utilise un initialisateur permettant d'associer un intercepteur à une application (i.e. client ou serveur). Cette association représente le lien qui est créé entre le méta-objet (i.e. intercepteur) et l'objet (i.e. l'application). Cette phase représente la phase d'initialisation du protocole à méta-objet (i.e. le lien entre le niveau de base et le méta-niveau).

C4 : Problème et incidence sur la conception de DAISY

Un ensemble d'intercepteurs ne peut être associé à une application qu'en série, ce qui nécessite la cohérence de leurs mécanismes non-fonctionnels. C'est pour cette raison que la coexistence de deux applications avec leur ensemble d'intercepteurs peut ne pas être possible.

Lors de l'exécution de l'application, le protocole à méta-objet assure l'observation et le contrôle d'un seul concept du modèle à objet CORBA, à savoir les requêtes.

Pour que les intercepteurs observent les requêtes, l'intergiciel applique un protocole simple qui se déroule de la manière suivante :

- Juste avant l'envoi de la requête, lorsque la méthode `downcall` au niveau du stub est activée, la méthode `send_request` ou `send_poll` de l'interface de l'intercepteur client est appelée. De la même manière, lorsque la requête arrive au niveau du squelette du serveur, et avant de la délivrer au serveur (i.e. avant l'activation de la méthode `upcall`), la méthode `receive_request_context` de l'interface de l'intercepteur serveur est activée. Ensuite, juste avant la transmission de la requête au

serveur, lors de l'activation de la méthode `upcall`, la méthode `receive_request` est appelée.

- Juste avant de retourner une réponse (i.e. appel de la méthode `downcall`) le squelette appelle les méthodes `send_reply`, `send_others` ou `send_exception` de l'interface de l'intercepteur serveur. La réponse destinée au client est réifiée à l'intercepteur client en activant les méthodes `receive_reply`, `receive_others` ou `receive_exception` lors de l'appel de la méthode `upcall` de la souche du client.

Le traitement des requêtes, effectué par l'intercepteur, peut être neutre. Ceci correspond au fonctionnement de l'intercepteur sans l'implémentation des méthodes de son interface. En revanche, le contrôle des requêtes peut être assuré par leur re-direction vers un autre serveur. Ceci est assuré par le déclenchement de l'exception `ForwardRequest`.

En résumé, le protocole défini par la norme CORBA :

i)- réifie toute requête issue d'un client à l'intercepteur de requête client *PIC*.

ii)- réifie toute requête à destination d'un serveur à l'intercepteur de requête serveur *PIS*.

iii)- délivre toutes les requêtes réifiées par les *PIC* et *PIS* au serveur cible; sauf dans le cas où l'intercepteur déclenche l'exception `Forward_Request`.

Les intercepteurs sont intimement liés aux applications auxquelles elles sont associées. La défaillance de l'application par arrêt entraîne l'arrêt de son intercepteur.

C5 : Problème et incidence sur la conception de DAISY

Le cycle de vie de l'intercepteur est lié à celui de l'application. C'est pour cette raison que la défaillance d'une application affecte une partie de l'ORB et des mécanismes non-fonctionnels qui y sont définis.

III.3.2.2. Prototype de la réplication passive

Nous présentons, Figure 37, l'architecture de la variante à base de méta-objets intégrés de la plate-forme DAISY. Dans cette architecture, nous traitons la mise en œuvre de la réplication passive.

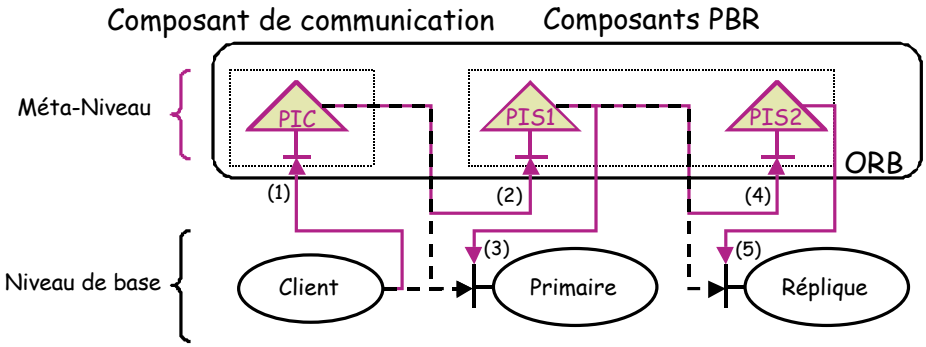


Figure 37. Réplication passive à méta-objets intégrés

Comme le montre cette figure, le méta-niveau renferme les composants de tolérance aux fautes, à savoir le composant de communication client/serveur et les composants PBR. Contrairement à la première variante de la plate-forme DAISY où les méta-objets sont des objets CORBA, dans cette seconde variante, les méta-objets sont des intercepteurs. Le composant *CC-C/S* est représenté par le *PIC* et les composants *PBR* par les *PIS1* et *PIS2*. Le niveau de base est composé d'un *Client* et de deux serveurs d'applications, le serveur *Primaire* et le serveur *Réplique*.

Lors de l'exécution de l'application, toute requête issue du client est réifiée par le PIC (1). Le composant de communication envoie ces requêtes au serveur primaire qui seront réifiées par le PIS1 (2). Ce méta-objet analyse les requêtes qui lui sont parvenues puis les transmet au serveur Primaire (3). Par ailleurs, le PIS1 récupère l'état du serveur primaire puis élabore une requête de mise à jour de l'état du serveur réplique. Cette requête est réifiée à l'intercepteur PIS2 (4) qui la traite, puis assure la mise à jour du serveur réplique (5).

Pour que les intercepteurs puissent envoyer des requêtes, ils récupèrent les IOR des différents objets du niveau de base. Ces identificateurs sont utilisés par la suite pour la mise en œuvre des mécanismes de réplication passive et non pas la mise en œuvre d'un protocole à méta-objet, comme cela était le cas pour la première variante de la plate-forme DAISY.

III.3.2.3. Prototype de la réplication active

Nous présentons, Figure 38, l'architecture de la variante à base de méta-objets intégrés de la plate-forme DAISY, dans laquelle nous traitons la mise en œuvre de la réplication active.

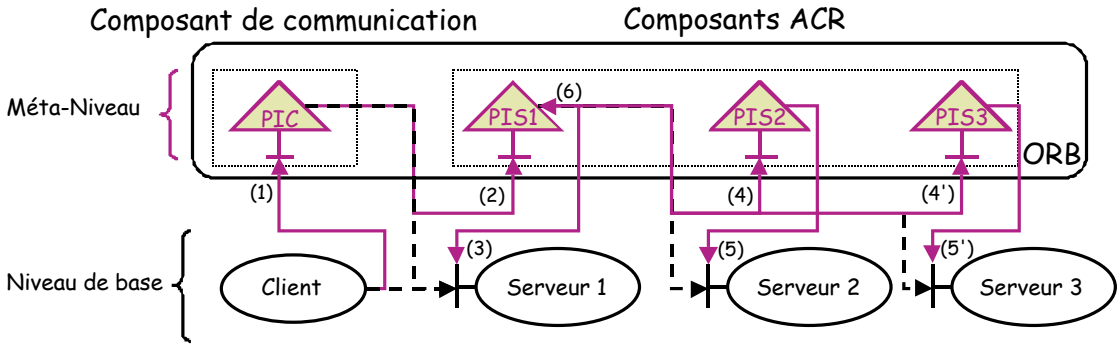


Figure 38. Réplication active à méta-objets intégrés

Comme le montre la Figure 38, le méta-niveau renferme les composants de tolérance aux fautes, à savoir les composants ACR et le composant de communication client/serveur. Le composant de communication est représenté par le *PIC* et les composants *ACR* se trouvent au niveau des intercepteurs *PIS1*, *PIS2* et *PIS3*. Le niveau de base est composé d'un client et de trois serveurs d'applications : le *Serveur 1*, le *Serveur 2* et le *Serveur 3*. Ces trois serveurs forment le groupe de répliques. En observant et en contrôlant le comportement et l'état des objets du niveau de base, les différents intercepteurs leur fournissent les mécanismes de la réplication active.

Lors de l'initialisation des objets du niveau de base, nous leur associons des intercepteurs qui jouent le rôle de méta-objets. L'intercepteur PIC qui représente le composant de communication client/serveur est associé à l'objet client. Les intercepteurs PIS1, PIS2 et PIS3 qui jouent le rôle des composants ACR sont associés aux serveurs Serveur 1, Serveur 2 et Serveur 3 respectivement. Cette phase représente la phase d'initialisation du protocole à méta-objet (i.e. le lien entre le niveau de base et le méta-niveau).

Lors de l'exécution de l'application, toute requête issue du client est réifiée au PIC (1) qui joue le rôle du composant CC-C/S. Ces requêtes qui sont transmises au Serveur 1 seront réifiées à l'intercepteur PIS1 (2). Ce méta-objet analyse les requêtes qui lui sont parvenues puis les transmet à l'objet du niveau de base qui lui est associé Serveur 1 (3). Pour assurer les mécanismes de réplication active, l'intercepteur PIS1 diffuse toutes les requêtes qui lui parviennent aux autres serveurs du groupe de répliques (i.e. Serveur 2 et Serveur 3). Ces requêtes sont réifiées aux intercepteurs PIS2 et PIS3 (4, 4'). Chaque méta-objet invoque par la suite l'objet du niveau de base qui lui est associé (5, 5'). Après avoir traité les requêtes, les serveurs retournent leurs réponses au méta-objet PIS1 (6) en passant par leurs méta-objets respectifs PIS2 et PIS3. Enfin, le PIS1 applique le vote et retourne la réponse majoritaire au client.

III.3.2.4. Classification

L'approche de mise en œuvre de la tolérance aux fautes de la variante à composants intégrés de DAISY est réflexive ; les critères relatifs à la classification des approches classiques, à savoir l'emplacement des mécanismes de tolérance aux fautes et leur forme d'activation, restent applicables pour ce type d'approche. En effet, les composants

responsables de la mise en œuvre des mécanismes de réplication (i.e. le méta-niveau) se trouvent intégrées dans l'intergiciel CORBA. En outre, tout client envoie sa requête sans être au courant de l'existence de cette plate-forme, et du fait qu'elle peut lui retourner une réponse même lors de la défaillance du serveur destinataire. Nous positionnons cette seconde variante dans la même classe que l'approche ORBIX+ISIS (voir Tableau 1 du chapitre II), c'est-à-dire parmi les approches par intégration et implicites.

Nous avons mentionné que les approches par intégration ne sont pas inter-opérationnelles à cause de l'intrusivité de l'approche par rapport à l'implémentation de l'intergiciel ; ceci aurait été un inconvénient majeur pour notre plate-forme. Cependant, la nouvelle norme standard de CORBA, permet l'introduction au sein de l'intergiciel du code relatif à des pré et post-traitements d'une requête en garantissant l'interopérabilité des applications. Ainsi, nous avons utilisé ce degré de liberté pour intégrer les mécanismes de réplication de services au sein de l'intergiciel.

L'objectif des approches implicites est de ne pas impliquer le développeur de l'application cliente dans le processus de génération du code relatif à la tolérance aux fautes. Comme la spécification de l'utilisation des instructions en pré et post-traitement se fait lors du lancement de l'application moyennant des paramètres d'initialisation, le développeur de l'application est par conséquent non responsable de l'activation des mécanismes de tolérance aux fautes. Ainsi, le code généré par le développeur peut fonctionner aussi bien dans un contexte simple que dans un contexte tolérant les fautes.

Selon la classification, concernant les approches réflexives, qui est définie dans le chapitre II, cette seconde variante de l'architecture de DAISY fait partie de la famille <A,I> (c'est-à-dire que le niveau de base se trouve dans la couche applicative et le méta-niveau se trouve au niveau de la couche intermédiaire). En effet, l'activité des différents composants de l'application, comme les serveurs et clients (i.e. niveau de base), est observée et gérée au niveau de l'intergiciel et plus particulièrement par les intercepteurs (i.e. méta-niveau) afin de fournir les mécanismes nécessaires pour la réplication de service.

Cette approche se distingue par le fait que le méta-niveau se trouve dans une couche du système informatique inférieure à celle du niveau de base. En effet, toutes les approches que nous avons présentées dans le second chapitre possèdent un méta-niveau qui se trouve dans la même couche ou dans une couche supérieure à celle du niveau de base.

Il est vrai que plus nous descendons dans les couches, plus nous gagnons en termes d'informations sur le fonctionnement du système ; cependant, nous perdons en termes de sémantique générale, d'où la difficulté qui se dresse devant cette seconde variante de DAISY. Pour pallier ce problème, et comprendre la signification de la sémantique des messages qui transitent via l'ORB, nous utilisons les méta-données qui sont ajoutées à tout message partant de la couche applicative vers la couche intergiciel (voir section 1). En conjuguant cette nouvelle potentialité avec les informations relatives au fonctionnement de l'intergiciel qui sont observables au niveau de cette couche, par exemple l'envoi et la réception de requêtes, cette approche se trouve avantagée par rapport aux approches antérieures. En effet, dans les

approches antérieures, la réification de telles informations par le méta-niveau n'est pas simple puisqu'elle fait appel à des techniques différentes, complexes et non portables, comme la compilation ouverte ou la transformation du code de l'application lors du chargement

III.3.3. Comparaison et choix entre les deux variantes

Les deux approches présentées dans les deux sections précédentes permettent de mettre en place les différents mécanismes réflexifs (la réification, l'intercession comportementale, l'introspection et l'intercession structurelle) avec pour chacune des avantages et des limites (voir Tableau 7).

Mécanismes réflexifs / DAISY	Réification	Intercession comportementale	Introspection	Intercession structurelle
À méta-objets externes	++	++	++	++
À méta-objets intégrés	+	+	++	++

Tableau 7. Les mécanismes réflexifs offerts par les deux versions de DAISY

Avec la première variante, le choix du méta-objet peut être effectué aussi bien lors du chargement de l'application que lors de son exécution. En outre, nous pouvons définir des règles de réification implémentant un filtrage par exemple. Ni le choix du méta-objet, ni même la définition de règles de réification n'est possible avec la seconde variante. La réification est assurée par l'intergiciel d'une manière statique et non modifiable (i.e. toutes les requêtes sont réifiées au niveau des intercepteurs).

Comme les intercepteurs ne peuvent que rediriger une requête vers un autre serveur en soulevant l'exception `ForwardRequest`, l'intercession comportementale offerte par la seconde variante est très limitée. En revanche, un méta-serveur, de la seconde variante, a la possibilité de changer les paramètres d'une invocation avant de l'appliquer au serveur du niveau de base. Aussi, il peut même invoquer une méthode différente de celle qui lui est réifiée au niveau du serveur d'application.

Par ailleurs, pour les mécanismes d'intercession structurelle et d'introspection, les deux versions sont équivalentes, puisque ces mécanismes sont dépendants des méta-interfaces offertes par les serveurs d'applications.

Les deux variantes permettent de mettre en œuvre les trois mécanismes de réplication classique. Seule la variante à composants intégrés, altère les mécanismes de la réplication active. En effet, dans le cas où le serveur associé au serveur qui diffuse la requête défaille en valeur, le *PIS* ne peut pas retourner une réponse qu'il a récupérée d'un autre serveur. Nous présentons en détail la modification de la réplication active dans le prochain chapitre.

L'utilisation d'un service de communication de groupe facilite la mise en œuvre de mécanismes de réplications. Munir la plate-forme DAISY d'un service de communication de groupe peut être envisagé dans la première variante puisque les méta-objets ne sont pas limités en termes de contrôle et d'observation et qu'ils possèdent des interfaces modifiables. En revanche, avec une interface non modifiable et un mode d'activation lié au flot des requêtes, les intercepteurs sont mal adaptés à la mise en œuvre d'un système de communication de groupe.

Nous avons choisi de développer la seconde variante de l'architecture de DAISY pour deux raisons :

Dans la première variante, les intercepteurs CORBA sont utilisés pour mettre en œuvre un protocole à méta-objets. Les mécanismes de réplication sont intégrés au niveau des méta-objets qui se trouvent au niveau applicatif. Ceci limite l'évaluation des intercepteurs par rapport à leur utilisation dans la mise en œuvre de la tolérance aux fautes. En revanche, dans la seconde variante, les intercepteurs CORBA jouent le rôle de méta-objets et intègrent les mécanismes de réplication. Ceci nous donnera plus d'éléments pour juger de la possibilité d'intégrer la tolérance aux fautes au niveau de la couche intermédiaire d'une manière standard.

Le potentiel de réification offert par le protocole à méta-objet à base d'intercepteurs CORBA, se limite à la re-direction d'une invocation à un méta-serveur ayant une interface équivalente à celle qui est proposée par le serveur de base. C'est-à-dire que l'interface du méta-serveur doit contenir les méthodes définies au niveau de l'interface du serveur de base. L'aspect non fonctionnel est, par conséquent, caché derrière cette interface. Ceci est très contraignant par rapport au développement des composants de tolérance aux fautes car ils doivent tenir compte des interfaces des applications sous-jacentes. En conséquence, cette architecture n'est pas réutilisable facilement.

III.3.4. Impact de l'utilisation de DAISY sur la conception des applications

Comme nous l'avons indiqué dans le paragraphe III.2 du chapitre 1, les interactions entre niveau de base (i.e. application) et le méta-niveau (i.e. les différentes composants de tolérance aux fautes) se font par le biais des méta-interfaces. Dans la plate-forme DAISY, chaque méta-objet possède une méta-interface qui observe et contrôle le comportement de l'application. Cependant, les interactions concernant la capture et la mise à jour de l'état de l'application ne sont pas fournies par les applications d'une manière standard à travers une méta-interface. Ainsi, pour pallier ce problème, nous avons conçu une classe dont toute application doit hériter. L'héritage de cette classe a pour rôle de munir l'application d'une méta-interface assurant l'aspect introspection (i.e. capture de l'état) et intercession structurelle (i.e. mise à jour de l'état).

La Figure 39 montre le diagramme de classe que doit avoir toute application.

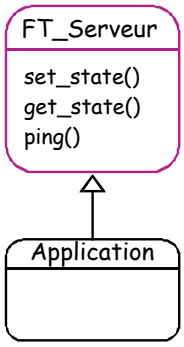


Figure 39. Diagramme de classe d'une application utilisant DAISY

En effet, outre les classes fonctionnelles propres à l'application, cette dernière doit hériter de la classe `FT_Server`. Cette classe fournit une interface renfermant trois méthodes qui sont responsables de la récupération et la mise à jour de l'état de l'application.

La méthode `get_state`, de la méta-interface de l'application, permet aux composants de tolérance aux fautes de la plate-forme DAISY de récupérer l'état interne du serveur indépendamment de l'application.

La méthode `set_state`, permet la mise à jour de l'état interne d'un serveur; la récupération et la mise à jour de l'état d'un serveur est rendue indépendante de l'application grâce à la sérialisation de l'état proposée par l'exécutif JAVA.

La troisième et dernière méthode `ping`, est utilisée par la plate-forme DAISY pour s'assurer de la non-défaillance d'une réplique; ceci renforce le pouvoir d'introspection de l'application par le méta-niveau.

III.4. LIMITES CONCEPTUELLES DES INTERCEPTEURS CORBA

En se basant sur les architectures définies précédemment, nous allons présenter dans cette section les limites conceptuelles des intercepteurs standard CORBA. Dans la première partie, nous présentons ces limites par rapport à leur utilisation pour construire un protocole à méta-objets, comme cela a été le cas pour la variante à composants externes de l'architecture de DAISY. Dans la seconde partie, nous nous focalisons sur les limites des intercepteurs par rapport à leur utilisation en tant que méta-objets pour la tolérance aux fautes ; cette partie est relative à la variante à composants intégrés de notre plate-forme.

III.4.1. Protocole à méta-objets à base d'intercepteurs CORBA

Nous montrons dans le Tableau 8 les limites des intercepteurs CORBA dans le cas où on les utilise comme composants de bases pour la mise en œuvre d'un protocole à méta-objets à l'exécution.

	Objet et interface	
Intercepteurs CORBA	N'assure pas l'observabilité	N'assure pas la contrôlabilité

Tableau 8. Limites des intercepteurs en tant que protocole à méta-objets

Ce type de MOP doit prendre en compte les différents éléments du concept à objet CORBA. Ainsi, il doit couvrir les notions d'objet, d'interface et de requête qui véhicule les interactions entre les objets CORBA. En utilisant les deux variantes d'intercepteurs de requêtes CORBA (i.e. *PIS* et *PIC*), un méta-objet peut observer et contrôler toutes les requêtes sortantes d'une application cliente ou à destination d'un serveur. En revanche, ces intercepteurs ne peuvent pas mettre en œuvre un lien entre méta-objet et objet pour pouvoir observer et contrôler ce dernier. Ces intercepteurs ne permettent que la gestion des requêtes CORBA.

III.4.1.1. Observabilité des objets et des interfaces

Les intercepteurs de requêtes ne permettent pas l'observation de l'objet et de son interface (i.e. introspection). En effet, pour que le MOP puisse prendre en compte ce type d'interactions, il faut que l'objet du niveau de base soit observable, c'est-à-dire qu'il offre une méta-interface renfermant des méthodes permettant l'observation de l'objet, ses attributs publics et privés et son interface.

L'intercepteur de requêtes permet de notifier le méta-objet lors de la création d'un objet du niveau de base, mais il est incapable de lui transmettre sa référence lors de cette notification. Le problème réside dans le fait que l'intercepteur est créé avant l'objet auquel il est associé. Ceci est très contraignant puisque le méta-objet doit attendre la première invocation adressée à cet objet pour que l'intercepteur lui transmette son IOR.

L'intercepteur de requêtes permet de notifier le méta-objet de la destruction d'un objet du niveau de base. En effet, la destruction d'un objet CORBA entraîne l'activation de la méthode `destroy` de l'intercepteur CORBA. En surchargeant cette méthode que l'intercepteur hérite de l'interface `PortableInterceptor::Interceptor`, ce dernier peut notifier le méta-objet de cet appel juste avant sa destruction.

Par ailleurs, la norme CORBA prévoit l'observation de l'interface d'un objet en utilisant le répertoire d'interface. Ce dernier stocke la description des interfaces sous la forme d'objets accessibles par le client et l'ORB.

III.4.1.2. Contrôlabilité des objets et des interfaces

En utilisant seulement les intercepteurs de requêtes CORBA, un méta-objet ne peut modifier ni la structure, ni l'état ni même l'interface d'un objet du niveau de base. En effet, pour que le MOP puisse prendre en compte ce type d'interactions, il faut que l'objet du niveau de base soit contrôlable, c'est-à-dire qu'il offre une méta-interface ou que l'ensemble de ces attributs soit déclaré au niveau de son interface fonctionnelle.

Pour permettre aux méta-objets de contrôler les objets du niveau de base dans l'architecture DAISY, aussi bien avec la première qu'avec la seconde variante, nous nous sommes basés sur une solution langage. Tout objet doit hériter de la classe *FT_Serveur* qui le munit d'une méta-interface. Cette dernière est prise en compte par le MOP pour assurer le lien de contrôle entre l'objet et son méta-objet (i.e. intercession structurelle).

Le seul aspect de contrôlabilité associé aux objets, pris en compte par la norme CORBA, est celui de la création des références d'objets IOR. Cet aspect de contrôle est assuré par les intercepteurs de références *IOR Interceptors* (voir Section 1).

III.4.2. Méta-objet de tolérance aux fautes à base d'intercepteurs CORBA

Nous montrons dans le Tableau 9 les limites des intercepteurs CORBA dans le cas où on les utilise comme des méta-objets pour mettre en œuvre des applications tolérants les fautes. Nous présentons ces limites aussi bien en termes de mécanismes réflexifs qu'en termes de mécanismes de tolérance aux fautes.

	Aspect Réflexif			Aspect Tolérance aux fautes	
Intercepteurs CORBA	Non circulaire	Absence de méta-interface adaptable	Association unique d'un ensemble d'intercepteurs	Élargissement de la zone de confinement des erreurs	Non évolutif

Tableau 9. Limites des intercepteurs en tant que méta-objets

Dans les systèmes réflexifs, l'aspect circulaire permet d'associer un méta-objet à tout objet ou méta-objet. Cette notion permet de munir une application de plusieurs méta-niveaux, c'est-à-dire de plusieurs mécanismes non-fonctionnels. La communication, entre les méta-objets du même niveau, de niveaux différents ou même avec des objets externes, se fait par l'intermédiaire des méta-interfaces. Les méta-interfaces sont indispensables pour que les méta-objets puissent communiquer ensemble. La prise en compte d'un ou plusieurs méta-objets dans un méta-niveau donné relève de la flexibilité du protocole à méta-objet qui les gère. Cette flexibilité, qui permet le changement de méta-objets, élargit le potentiel des solutions architecturales.

Dans les systèmes critiques, plus la zone de confinement d'erreur est bien isolée plus les effets de corrélation des erreurs sont limités. Comme ces systèmes sont souvent des systèmes à longue durée de vie, les besoins en termes de tolérance aux fautes peuvent évoluer. Par conséquent, l'ajout ou la modification de ces fonctionnalités orthogonales par le biais des intercepteurs CORBA doivent être dynamiques et indépendants des applications, et ce pour des raisons de disponibilité.

III.4.2.1. Absence de méta-interface adaptable

La norme CORBA ne prévoit que deux interfaces standard pour les différents types d'intercepteurs, une première interface pour les intercepteurs du côté client *PIC* et une seconde pour ceux qui se trouvent du côté serveur *PIS*. L'activation de ces méthodes est assurée par un pseudo-protocole à méta-objets fourni par l'intergiciel ; nous ne pouvons ni ajouter de méthodes à l'interface fournie par les intercepteurs CORBA ni même les activer d'une manière directe.

Nous avons pu résoudre ces deux problèmes dans le cas des PIS. En effet, pour ajouter de nouvelles méthodes au niveau de la méta-interface de l'intercepteur, nous associons une classe de tolérance aux fautes, par héritage, à la classe principale de toute application. L'implémentation de ces méthodes se trouve au niveau des intercepteurs et non pas au niveau du serveur de base qui doit, quant à lui, uniquement fournir une implémentation vide de ces méthodes. Par ailleurs, l'activation de ces méta-méthodes est similaire à l'activation d'une méthode d'un serveur de base. Le seul inconvénient est que ces méthodes ne peuvent pas retourner de réponse.

En revanche, la solution proposée pour les PIS n'est pas envisageable pour les PIC car le client ne possède pas d'interface et l'intercepteur client n'est pas adressable; seule la communication par exception est possible. En effet, pour que le PIS communique avec le PIC, l'intercepteur serveur doit générer une exception qui sera interprétée par le PIC lorsqu'il l'intercepte, juste avant qu'il ne la transmette au client. Ce mode de communication est très restreint, il limite les interactions entre les différents objets du méta-niveau (i.e. PIC et PIS). Par ailleurs, selon la norme CORBA, l'activation de la méthode `receive_exception` d'un intercepteur engendre l'activation systématique de la même méthode des autres intercepteurs. Ainsi, même si nous voulons communiquer par exception, nous n'avons pas le pouvoir de cibler un seul intercepteur.

III.4.2.2. Intercepteur non circulaire

Comme le montre la Figure 40, les intercepteurs CORBA peuvent être disposés en étages.

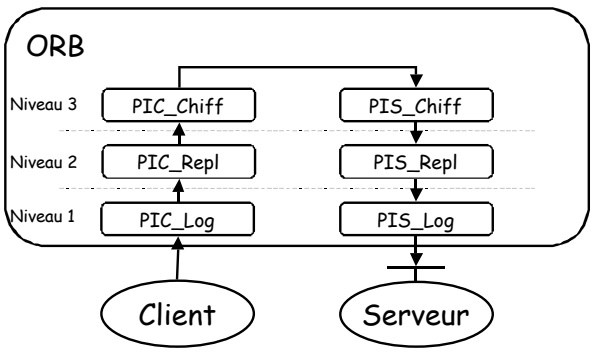


Figure 40. Intercepteurs CORBA en étages

Cette disposition est appropriée pour des traitements complémentaires. Par exemple, si nous nous proposons de concevoir un système qui tient un journal des requêtes qu'il a traitées, qui soit répliqué et qui soit sécurisé, une solution à trois étages d'intercepteurs peut être envisagée. En effet, le premier étage, formé par deux intercepteurs, enregistre toutes les requêtes sortantes d'un client *PIC_Log* ainsi que celles qui sont rentrantes à un serveur *PIS_Log*. Le second étage assure la réplication du serveur aux clients qui l'utilisent, *PIC_Repl et PIS_Repl*. Quant au troisième étage, il est responsable du chiffrement, côté client *PIC_Chiff*, ainsi que du déchiffrement, côté serveur *PIS_Chiff*, des requêtes.

La disposition proposée dans la Figure 40 est similaire à celle proposée par Smith [Smith 1982] et Fabre [Fabre 1998]. L'approche proposée par Smith se base sur la notion d'interpréteurs méta-circulaires. Un interpréteur méta-circulaire donne accès à une représentation (méta-modèle) de son propre processus d'interprétation, et permet, en modifiant cette représentation, de modifier son propre fonctionnement. L'approche proposée dans [Fabre 1998] est appliquée aux objets. Ainsi, un méta-objet, qui influence le fonctionnement d'un objet du niveau de base, peut être observé et contrôlé par un méta-méta-objet qui conditionne son propre fonctionnement.

Malgré le fait que les intercepteurs CORBA soient des objets, il nous est impossible, selon la norme CORBA, de leur associer des intercepteurs. Par exemple, le PIC_Repl de la Figure 40 n'est pas le méta-objet du PIC_Log. Il est un second méta-objet du client qui assure le traitement des requêtes sortantes après le PIS_Log. Ceci limite l'aspect réflexif au niveau de l'intergiciel. En effet, une fois que l'intercepteur est déployé, nous ne pouvons ni observer, ni contrôler son comportement. Voici donc une autre limite des intercepteurs en terme de réflexivité : on ne peut leur associer de méta-niveau.

III.4.2.3. Association unique d'un ensemble d'intercepteurs CORBA

La norme CORBA ne prévoit que l'association en série des intercepteurs ; l'association de deux intercepteurs à une application ne peut être faite d'une manière exclusive. En effet, une fois que les intercepteurs que nous allons utiliser dans une architecture donnée sont sélectionnés, ils intercepteront les requêtes d'une manière successive, selon l'ordre de leur définition. Nous ne pouvons pas choisir parmi ces intercepteurs, un sous-ensemble qui va réifier les requêtes. En ne permettant qu'une seule façon d'associer les intercepteurs aux applications (en série), la norme CORBA rend l'utilisation des intercepteurs très contraignante et peut même avoir une incidence sur la conception des systèmes.

A titre d'exemple, dans la Figure 41 nous associons aux applications *A1* et *A2* des mécanismes de réplication avec DAISY.

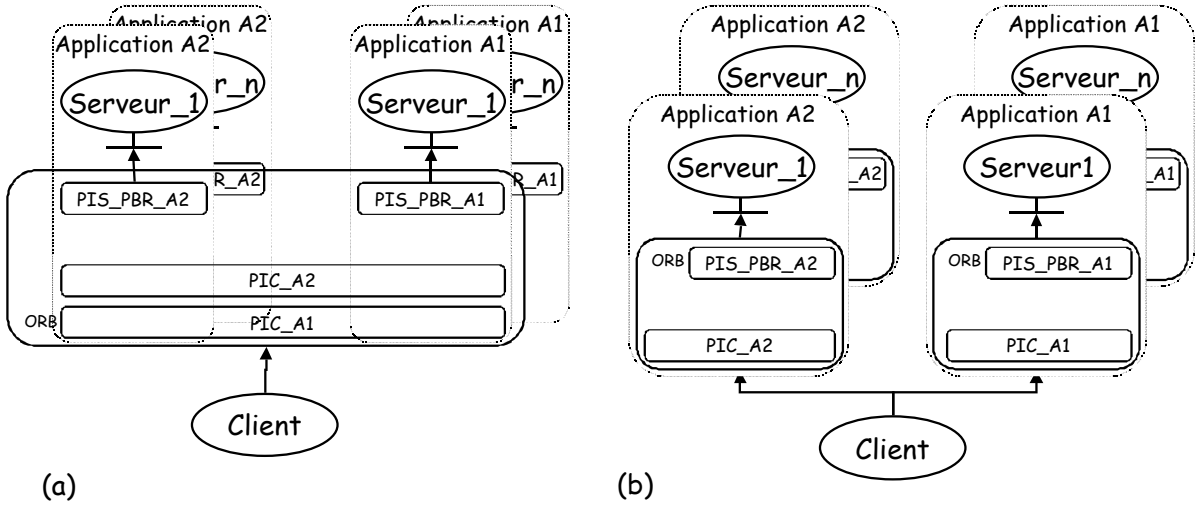


Figure 41. Association (Série/Parallèle) des intercepteurs

Un client voulant utiliser ces deux applications, doit initialiser deux composants de communication client/serveur *CC-C/S* (i.e. *PIC_A1* et *PIC_A2*). Par ailleurs, chaque serveur d'application (i.e. *Serveur_1*, *Serveur_n*) est muni d'un composant de réplication passive *PBR* (i.e. *PIS_PBR_A1* et *PIS_PBR_A2*). Comme le montre la figure (a), un client voulant demander un premier service de l'application *A1* puis un second service de l'application *A2* voit sa requête interceptée dans les deux cas par le composant de communication client/serveur *PIC* de l'application qui n'est pas concernée. Cette disposition peut induire des défaillances, suite à la corrélation des erreurs, dans la mesure où la détection d'une défaillance par un intercepteur entraîne, selon la norme CORBA, le déclenchement du mécanisme de détection de défaillance de l'intercepteur qui le suit. Une association en parallèle (voir figure b) permettrait au client d'avoir deux composants de communication client/serveur distincts, ce qui limiterait les interférences entre les différents mécanismes.

L'utilisation forcée d'une telle association rend le système plus lent, puisqu'une requête doit passer par tout l'ensemble des intercepteurs, même s'il y en a une partie qui n'est pas concernée par celle-ci. En outre, la réification des requêtes par ce sous-ensemble d'intercepteurs fragilise le système et le rend plus vulnérable, puisqu'il augmente les effets de bords qui peuvent apparaître lors des interactions entre des niveaux qui ne peuvent pas coexister.

En somme, ce problème, que nous avons relevé dans le cadre d'une application de tolérance aux fautes, se pose pour tous les types de mécanismes non-fonctionnels. En effet, l'intégration de plusieurs types de ces mécanismes peut s'avérer non seulement difficile mais aussi contraignante car elle peut nécessiter la révision de la phase de conception du système entier.

III.4.2.4. Élargissement de la zone de confinement des erreurs

La norme CORBA spécifie que la destruction d'une application entraîne la destruction de l'intercepteur auquel elle est associée. Cette association nous semble rigide dans la mesure où la défaillance d'une application entraîne inéluctablement celle de son méta-objet. Cette

rigidité élargit la zone de propagation d'erreurs d'une part, et ne peut pas aider à la confiner d'autre part.

Si l'intercepteur était lié au serveur d'application d'une manière moins rigide, l'erreur qui a engendré la défaillance du premier aurait été confinée à l'application et non pas à l'ensemble formé par l'application et son intercepteur. Cette zone s'élargit encore plus dans le cas où l'application est munie de plusieurs intercepteurs.

Cette rigidité du lien entre l'application et son méta-objet empêche ce dernier de relever la défaillance de l'application et de mettre en œuvre des mécanismes pour arrêter la propagation de l'erreur puisqu'il est détruit au même moment. Par conséquent, un méta-objet ne peut pas confiner la zone de propagation de l'erreur de l'application. En outre, cette contrainte peut influencer la mise en œuvre d'autres mécanismes de tolérance aux fautes, à savoir l'empaquetage (i.e. *wrapping*) qui ne doit pas s'insérer entre les intercepteurs et leur serveur associé.

En effet, la barrière qui limite la propagation des erreurs lors de la défaillance d'un serveur s'étend jusqu'à la couche commune alors que dans le cas où l'intercepteur est moins lié à l'application, la détection de la défaillance d'un serveur sera confinée à la couche PBR ou LF ce qui limite le degré de propagation de la défaillance.

Par ailleurs, cette association introduit des contraintes de modularité pour la conception de plate-formes réflexives. En effet, en ayant des objets du niveau de bases qui ont le contrôle du cycle de vie des méta-objets, ceci rend l'ensemble formé par l'intercepteur et son application inséparables par rapport à la création et à la destruction.

III.4.2.5. Les intercepteurs entravent l'évolution des applications

Tels que définis par l'OMG, les intercepteurs ne peuvent pas être insérés ni retirés, dynamiquement à une application en cours d'exécution. Leur utilisation est spécifiée d'une manière définitive lors du lancement de l'application. L'insertion non dynamique des intercepteurs pose un problème de disponibilité et d'évolutivité pour les applications.

En effet, toutes les applications qui veulent introduire un nouvel aspect non-fonctionnel, par exemple, le traçage des messages ou d'autres mécanismes visant à faire évoluer le système entier, sont obligées de s'arrêter puis de redémarrer en spécifiant l'association des nouveaux intercepteurs, ce qui influence la disponibilité du service rendu par l'application.

Pour pouvoir simuler l'insertion dynamique d'un intercepteur, nous pouvons associer lors de l'initialisation de l'application plusieurs intercepteurs dont nous n'utilisons qu'une partie. Le détail de cette solution sera abordé dans le chapitre suivant. Malgré que cette solution semble résoudre cet inconvénient, elle ne peut pas résoudre le problème d'insertion de méta-mécanismes qui n'ont pas été pris en compte initialement.

III.5. CONCLUSION ET PRECONISATION DE NOUVELLES SOLUTIONS

Nous avons présenté, dans ce chapitre, les limites que posent les intercepteurs CORBA standards pour le développement d'architectures tolérants les fautes et réflexives. Nous avons discuté ces limites selon deux points de vue différents : d'une part pour construire un protocole à méta-objet à l'exécution et d'autre part, comme des méta-objets de tolérance aux fautes.

Pour le premier aspect, nous avons montré que les intercepteurs CORBA assurent :

- la réification de la création et de la destruction d'un objet ainsi que les interactions entre objets, et
- l'intercession structurelle qui permet la communication entre méta-objet et objet.

En revanche, ces intercepteurs ne peuvent pas fournir l'observation et le contrôle de l'état des objets (i.e. introspection et intercession structurelle).

Pour le second aspect, où nous considérons les intercepteurs comme des méta-objets de tolérance aux fautes, nous avons distingué deux types de limites : celles qui sont relatives à la réflexivité et celles qui sont relatives à la tolérance aux fautes.

Sur le plan de la réflexivité, nous avons montré la pauvreté de la méta-interface offerte par ces intercepteurs, leur affectation statique aux applications et leur incapacité d'être réflexif.

Sur le plan de la tolérance aux fautes, nous avons démontré que leur utilisation introduirait un élargissement de la zone de confinement des erreurs à l'espace contenant l'application et ses intercepteurs associés.

Pour pallier ces problèmes, nous proposons une amélioration des intercepteurs CORBA que nous présentons dans la Figure 42.

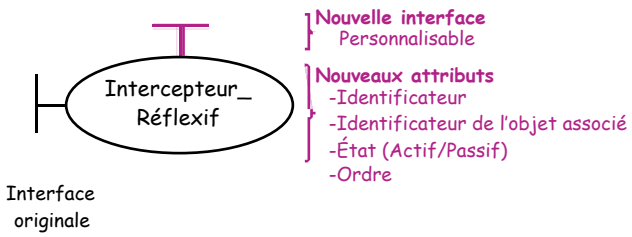


Figure 42. Nouvelle Génération d'intercepteurs

Le nouvel intercepteur se présente sous la forme d'un objet CORBA particulier et possède l'interface originale des intercepteurs actuels tels que spécifié par l'OMG. En outre, il offre une nouvelle méta-interface, personnalisable, qui lui permet d'offrir d'autres méta-services.

Cette interface est invocable par tous les objets CORBA ainsi que les intercepteurs, en utilisant l'identificateur de ce nouvel intercepteur. Il offre, par ailleurs, d'autres attributs comme l'identificateur de l'objet associé, qui permet de spécifier l'objet sur lequel se fait l'interception, l'état qui permet d'activer ou de neutraliser l'interface originale de l'intercepteur, et la notion d'ordre qui fournit plus de liberté lors de l'insertion d'un nouvel intercepteur.

Par ailleurs, nous présentons dans ce qui suit, les différentes solutions que l'intercepteur amélioré permet d'apporter aux problèmes décrits dans les deux sections précédentes.

Solution 1 : Intercepteur circulaire

Lorsque l'intercepteur est doté d'une nouvelle interface personnalisable, il est possible de lui associer un intercepteur qui pourra l'observer et le contrôler (i.e. méta-intercepteur). Ceci permettrait, par exemple, de doter un système de différents mécanismes non-fontionnels.

Solution 2 : Interface adaptable

L'utilisation d'une méta-interface personnalisable et d'un identificateur qui permet de repérer l'intercepteur cible, facilite la communication entre les intercepteurs. En fournissant une méta-méthode d'activation, par exemple, l'intercepteur client pourra être activé ou désactivé sur une simple invocation. Par ailleurs, nous n'aurons plus besoin d'obliger le concepteur d'une application à hériter d'une interface non-fonctionnelle pour permettre la communication avec les intercepteurs.

Solution 3 : Flexibilité de l'association d'un ensemble d'intercepteurs

Le choix des valeurs des attributs *état* et *ordre*, rend le chemin d'interception d'une invocation plus flexible. En effet, lorsqu'une application est initialisée avec des intercepteurs dont l'attribut *état* est activé, nous pouvons changer l'ordre d'activation de ces intercepteurs en changeant la valeur de l'attribut *ordre*. En outre, le changement de la variable *état* en inactif exclut l'intercepteur du chemin de la requête, c'est-à-dire qu'il n'intercepte plus de messages. Cette dernière configuration nous permet d'associer aux applications des intercepteurs en parallèle. Par ailleurs, nous pouvons retrouver le comportement actuel des intercepteurs CORBA en associant les intercepteurs à une application donnée en les initialisant avec un état actif et un ordre bien défini.

Solution 4 : Confiner la zone de propagation des erreurs

Avec un identificateur propre, l'intercepteur aura une identité différente de l'application à laquelle il est associé. De plus, la définition de l'attribut *identificateur de l'objet associé*, n'associe pas d'une manière définitive un intercepteur à une application donnée. Le problème relatif à la destruction conjointe de l'application et de son intercepteur sera pallié. En effet, lors de la destruction de l'application, seul l'attribut qui pointe sur l'objet associé sera affecté. Cette séparation entre l'intercepteur et l'application renforcera les capacités du premier qui

pourra, à cet effet, superviser même la création et la destruction de l'application, chose que l'actuelle version des intercepteurs ne permet pas. Par ailleurs, en termes de tolérance aux fautes, cette séparation limitera la propagation de l'erreur activée lors de la défaillance d'une application à l'intercepteur qui lui est associé.

Solution 5 : Insertion dynamique des intercepteurs

Il existe deux cas pour l'insertion dynamique des intercepteurs. Le premier, lorsque l'intercepteur est déclaré lors du lancement d'une application (i.e. client et/ ou serveur) avec un état *inactif*. L'insertion de cet intercepteur se fera sur un simple appel pour le changement de la valeur de l'attribut état à *actif*. En revanche, pour le second cas où l'intercepteur doit être inséré lorsque l'application est déjà en fonction, nous utiliserons l'identificateur de l'objet associé et l'ordre dans lequel cet intercepteur doit être activé pour pouvoir l'insérer dans le chemin de la requête.

Pour que ces améliorations puissent être prises en compte, il faut que le protocole à méta-objets qui gère les liens entre intercepteur et objet de base soit révisé.

CHAPITRE IV

MISE EN ŒUVRE DES MECANISMES REFLEXIFS STANDARDS ET ANALYSE

Dans ce chapitre, nous présentons l'implémentation de l'architecture de DAISY. Cette implémentation a pour objectifs de prouver que la mise en œuvre d'une telle architecture est possible en utilisant les intercepteurs CORBA en relevant et en palliant dans une certaine mesure les limites de ces derniers pour implémenter les mécanismes de tolérance aux fautes.

La première partie de ce chapitre est consacrée à l'implémentation de la réplication passive. Nous y détaillons les différents mécanismes que nous avons intégrés au niveau du composant de communication client/serveur et du composant de réplication passive. Les mécanismes relatifs à la réplication active font l'objet de la seconde partie. Dans ces deux premières parties, nous relevons les limites des intercepteurs actuels qui ont conditionné notre implémentation. Dans la troisième partie, nous présentons l'implémentation des composants utilitaires, à savoir l'usine à objets et le gestionnaire de réplication. Dans la dernière partie, nous discutons les limites des intercepteurs CORBA que nous avons dégagés lors du développement de ces deux modes de réplication. Nous terminons par une synthèse des résultats que nous avons obtenus lors de l'expérimentation de la plate-forme DAISY.

IV.1. MISE EN ŒUVRE DE LA REPLICATION PASSIVE

Dans l'implémentation des mécanismes de la réplication passive à l'aide des intercepteurs CORBA, les mécanismes sont définis au niveau des intercepteurs ; par conséquent, nous nous plaçons dans le cadre d'une architecture à méta-objets intégrés [Bennani 2004]. Cette implémentation, basée sur des COTS standards, concerne deux des composants de tolérance aux fautes de l'architecture de DAISY présentés dans la section 2 du chapitre III, à savoir le composant *CC-C/S* et le composant *PBR*. Le premier a pour rôle de masquer aux clients l'occurrence de la défaillance des serveurs aux clients; par exemple, lors de l'occurrence d'une défaillance du serveur primaire, le serveur secondaire sera promu serveur primaire de façon transparente pour le client. Le second se charge de la mise à jour de l'état des répliques et du changement du mode de fonctionnement d'un serveur.

IV.1.1. Initialisation de la plate-forme

À l'initialisation d'un serveur d'application, et dans le cas de la réplication passive, la plate-forme DAISY instancie plusieurs serveurs d'applications et associe à chacun d'entre eux un intercepteur serveur de réplication passive *PIS_PBR*. Un seul parmi l'ensemble de ces serveurs est initialisé en tant que serveur primaire, les autres sont des serveurs répliques. Cette différence ne réside pas au niveau du code de l'application, mais elle est liée au type de l'intercepteur serveur qui lui est associé. En effet, si la plate-forme associe l'intercepteur *PIS_PBRp* à l'application, le serveur sera primaire. En revanche, si elle lui associe l'intercepteur *PIS_PBRb*, le serveur sera une réplique.

À l'initialisation de toute application cliente, notre plate-forme lui associe le composant *CC-C/S* qui n'est autre qu'un intercepteur client *PIC*. Cet intercepteur n'est pas propre à la mise en œuvre de la réplication passive, il sera utilisé aussi pour la mise en œuvre de la réplication active.

IV.1.2. Intercepteur client

L'intercepteur client a pour charge d'identifier d'une manière unique le client qui lui est associé ainsi que toutes les requêtes qu'il émet, de détecter les défaillances du serveur primaire et de les masquer au client.

IV.1.2.1. Initialisation

Lors de l'initialisation du *PIC* par l'ORB, le constructeur de l'intercepteur génère un identificateur unique pour le client qui lui est associé et récupère les identificateurs *IOR* des différents serveurs appartenant au groupe de répliques.

L'identificateur unique du client se compose de l'adresse IP de la machine hôte et d'un numéro de séquence. Le premier champ est utilisé pour distinguer les clients qui résident sur des machines différentes, et est récupéré en utilisant la méthode `java.net.InetAddress.get_AllByName`. Le second sert à différencier les clients qui résident sur la même machine. Le numéro de séquence du client est déterminé à partir du fichier *client_repository*, qui représente un support de stockage pour les identificateurs des différents clients résidant de la même machine.

Les identificateurs *IOR* du serveur primaire et des serveurs répliques sont utilisés pour repérer les serveurs auxquels les requêtes seront redirigées. La re-direction des requêtes permet de masquer la défaillance, transitoire ou permanente, du serveur primaire. Une fois l'intercepteur créé, le client termine son instanciation et peut ainsi commencer à envoyer des requêtes.

IV.1.2.2. Comportement

Lorsque le client est fonctionnel, c'est-à-dire l'intercepteur qui lui est associé est initialisé, les mécanismes implémentés au niveau du "*PIC*" sont activés lors de l'interception

des évènements suivants : envoi d'une requête par le client, retour d'une réponse par le serveur primaire ou retour d'une exception (voir Figure 43).

```
1.  switch interception
2.  case request_out :                          {interception d'une requête}
3.      num_req := compt_req;                   {récupération de l'id de la requête}
4.      serialiser num_req;                     {mise en forme de l'id de la requête}
5.      serialiser id_client;                   {mise en forme de l'id du client}
6.      id_req := id_client + num_req;          {élaboration de l'id unique de la requête}
7.      piggyback id_req;
8.  break;
9.  case response :                             {interception d'une réponse}
10.     compt_req := compt_req + 1;             {mise à jour du compteur des requêtes}
11. break;
12. case exception :                           {interception d'une exception}
13.      num_essai := num_essai + 1;           {mise à jour du nombre de ré-émissions}
14.      si num_essai < N                      {défaillance transitoire}
15.          forward_request(primaire);        {ré-émission de la requête}
16.      sinon                                 {défaillance permanente}
17.          forward_request(replique_1);      {re-direction de la requête}
18.      finsi;
19. break;
20. fin switch;
```

Figure 43. Les mécanismes de réplication passive associés au PIC

Toute requête du client à destination du serveur primaire est interceptée par le *PIC*. Cette interception active le mécanisme d'identification des requêtes. En effet, l'intercepteur client commence par donner un numéro de séquence à cette requête pour pouvoir identifier chacune des requêtes sortantes d'un même client. Ensuite, il regroupe le numéro de séquence de la requête et l'identificateur du client afin de générer un identificateur unique pour la requête. Cet identificateur permet de différencier les requêtes qui sont émises par des clients différents. Enfin, il accroche (*piggyback*) cet identificateur au contenu de la requête avant de la délivrer au serveur primaire. L'identificateur accroché à la requête (i.e. méta-donnée) sera utilisé par les intercepteurs serveurs lors de la reprise. L'identificateur de requête correspond à un compteur qui est incrémenté lors de l'interception de la réponse issue du traitement de la requête.

Lors de l'occurrence d'une défaillance, le *PIC* intercepte l'exception retournée et la traite sans la transmettre au client. Nous distinguons deux types de défaillances : les défaillances transitoires[13] et les défaillances permanentes.

[13] L'erreur transitoire est différente de l'exception TRANSIENT de CORBA car l'exception TRANSIENT de CORBA correspond à une erreur de communication transitoire qui est détectée au bout de six minutes environ.

La défaillance transitoire est due à un échec de communication entre le client et le serveur primaire. Pour traiter ce type d'erreurs, l'intercepteur client ré-émet la requête au serveur primaire en utilisant son identificateur *IOR*, récupéré lors de son initialisation.

La défaillance permanente peut correspondre soit à arrêt inopiné du serveur, soit à une défaillance transitoire persistante. En effet, après *N* échecs de ré-émission de la requête par l'intercepteur client au serveur primaire, l'hypothèse de l'échec de la communication n'est plus valable. Dans les deux cas, nous considérons que la défaillance provient de l'arrêt du serveur primaire. En conséquence, l'intercepteur client re-dirige la requête vers la première réplique.

La première réplique est repérée par le nom *Replique_1* dans le serveur de désignation. Ainsi, le choix du serveur secondaire qui va remplacer le serveur primaire défaillant est fixé à priori.

Comme nous l'avons décrit précédemment, le client n'est pas impliqué dans la mise en œuvre des mécanismes de la réplication passive. En effet, il ne se charge pas de ré-envoyer sa requête vers un autre serveur puisque c'est le composant CC/C-S qui l'assure, et le client ne s'aperçoit même pas de la défaillance du primaire.

IV.1.3. Intercepteur primaire

L'intercepteur primaire *PIS_PBRp* est responsable de la génération des points de reprise et de la mise à jour, périodique, de l'état des différentes répliques (i.e. réplication passive à chaud). Le serveur primaire et son intercepteur sont liés de telle sorte que l'arrêt inopiné de l'un implique celui de l'autre. L'intercepteur primaire ne peut donc tolérer que l'arrêt inopiné des serveurs secondaires.

IV.1.3.1. Initialisation

Lors de l'initialisation de la plate-forme DAISY avec la réplication passive, les différents serveurs d'application utilisent la méthode `subscribe`, héritée de la classe `FT_Serveur` (voir la section 3 du chapitre III), pour s'enregistrer en tant que primaire ou comme réplique auprès du service de désignation de CORBA.

Dans le cas où l'application s'enregistre comme primaire, nous lui associons un intercepteur primaire *PIS_PBRp*. Ce dernier récupère les *IOR* des différents serveurs lancés avec le profil réplique lors de son initialisation. Ces identificateurs sont utilisés, lors des traitements, pour envoyer les mises à jour périodiques. Lorsque le *PIS_PBRp* et le serveur qui lui est associé sont bien initialisés, ils se mettent en attente des requêtes des clients.

Comme l'identificateur d'un serveur est généré après l'initialisation de l'intercepteur qui lui est associé, ce dernier récupère l'IOR du serveur primaire lors de l'interception de la première invocation. Cet identificateur est utilisé pour récupérer l'état du serveur primaire.

PI : Problème et incidence sur l'implémentation

L'intercepteur est incapable de connaître l'objet auquel est associé lors de son initialisation. C'est pour cette raison que dans notre implémentation, on doit attendre la première requête pour pouvoir le récupérer.

IV.1.3.2. Comportement et traitements

La Figure 44 montre le comportement de l'intercepteur *PIS_PBRp* lors de l'interception d'une requête à destination du serveur ou lors du retour d'une réponse.

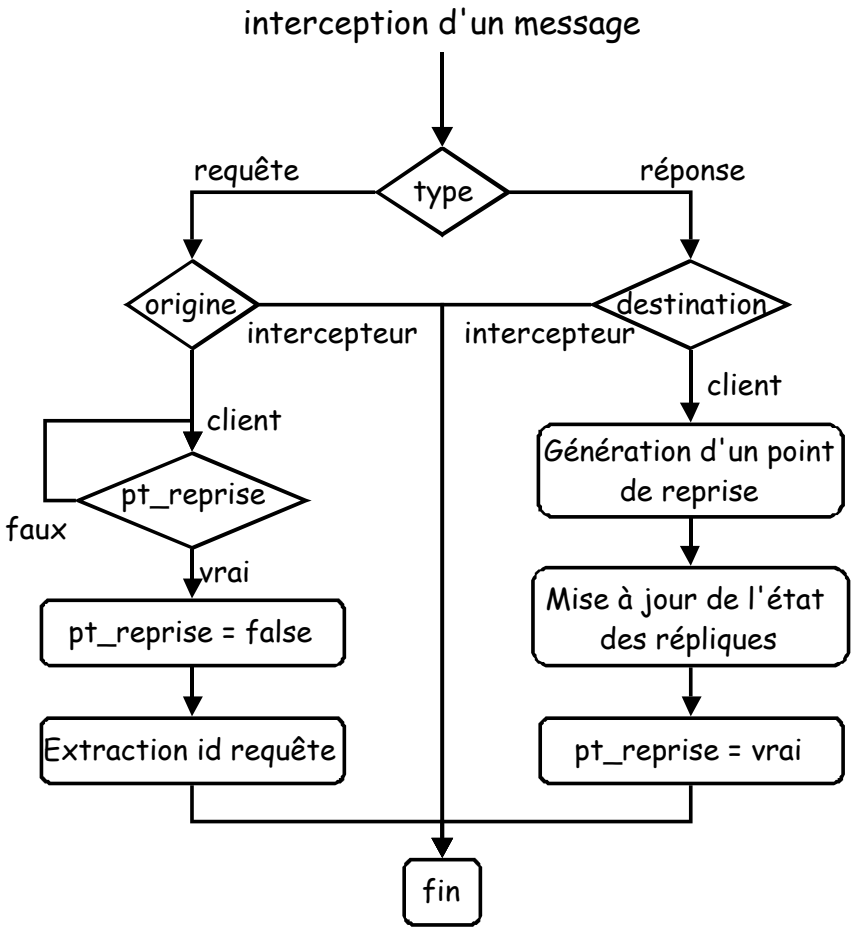


Figure 44. Capture et mise à jour de l'état des répliques

Le mécanisme de capture d'un point de reprise est activé lors de l'interception d'une réponse à destination d'un client. En effet, nous avons considéré qu'un serveur change d'état après avoir traité une requête fonctionnelle et généré une réponse. Nous identifions une réponse à destination d'un client à partir du champ `method` du paramètre `requestinfo` de la méthode `send_reply()` de l'intercepteur *PIS_PBRp* (voir la section 1 du chapitre III). Pour capturer un point de reprise, l'intercepteur primaire envoie la requête `get_state()` à destination du serveur primaire en utilisant son IOR. Cette méthode qui est définie dans la classe `FT_Serveur` est héritée par l'application et elle utilise le mécanisme standard de sérialisation de l'état fourni par la machine virtuelle JAVA. Lorsque l'intercepteur primaire est

en train de générer un point de reprise, toute requête émise par les clients à destination du serveur primaire est interceptée et bloquée. Ceci permet la prise d'un point de reprise atomique.

Dans l'implémentation, nous utilisons une variable particulière (la variable d'état *pt_reprise*) permettant de modéliser et de contrôler l'état de l'intercepteur : état de capture de point de reprise et état débloquant. Lorsque cette variable possède la valeur *faux*, elle indique le blocage des requêtes interceptées.

Suite à la récupération de l'état du serveur primaire, l'intercepteur *PIS_PBRp* utilise la méthode CORBA `set_state()` des composants *PIS_PBRb* pour mettre à jour l'état des différents serveurs secondaires. L'état est repéré avec l'identificateur de la requête qui l'a généré. L'intercepteur primaire utilise les IOR des différents serveurs secondaires pour activer leurs mises à jour. Enfin, il modifie la valeur de la variable d'état *pt_reprise* pour libérer l'une des requêtes mises en attente par notre protocole.

Puisque l'intercepteur doit laisser passer les requêtes d'une manière séquentielle, nous protégeons l'accès en lecture à la variable d'état *pt_reprise* par un sémaphore afin de garantir le déblocage d'une seule requête mise en attente. En effet, comme l'intercepteur hérite le brin d'exécution de la requête véhiculée par l'ORB, nous pouvons avoir un accès simultané pour lire le contenu de la variable d'état *pt_reprise* par plusieurs requêtes.

P2 : Problème et incidence sur l'implémentation

L'intercepteur ne possède pas de politique d'ordonnancement des messages. C'est pour cette raison que dans notre implémentation, l'ordre de traitement des messages peut être différent de l'ordre de leur interception.

L'intercepteur *PIS_PBRp* intercepte la requête `get_state()` qu'il a lui-même générée pour récupérer un point de reprise. Cette requête est délivrée au serveur primaire sans aucun pré-traitement. Elle est distinguée des autres requêtes « fonctionnelles » afin d'éviter son blocage par le protocole de traitement d'un point de reprise, ce qui bloquerait l'évolution de l'application.

P3 : Problème et incidence sur l'implémentation

L'intercepteur est une source de blocage. C'est pour cette raison que dans notre implémentation, on doit distinguer les requêtes générées par l'intercepteur de celles émises par des clients, car ces dernières sont systématiquement bloquées par les mécanismes de réplication au niveau de l'intercepteur.

Par ailleurs, lorsque l'intercepteur primaire intercepte une réponse dont il est la destination, par exemple la réponse de `get_state()`, il ne lui applique aucun post-traitement. Ces réponses sont distinguées de celles qui sont à destination des clients afin d'éviter une boucle

infinie. En effet, si ces réponses n'étaient pas traitées à part, le mécanisme de génération de point de reprise serait enclenché à nouveau.

Problème et incidence sur l'implémentation P4 :

L'intercepteur est une source de boucles infinies. C'est pour cette raison que dans notre implémentation, on doit distinguer les réponses à destination de l'intercepteur de celles destinées aux clients, car ces dernières génèrent des requêtes lors de leur interception.

IV.1.4. Intercepteur réplique

L'intercepteur réplique *PIS_PBRb* (i.e. l'intercepteur *backup*), associé à une application *Serveur*, a pour charge de gérer la mise à jour de l'état du serveur réplique (demandée par le *PIS_PBR*), de détecter (en collaboration avec les *PIC*) et vérifier la défaillance du serveur primaire et de gérer la reprise.

IV.1.4.1. Modélisation et Initialisation

Comme le montre la Figure 45, l'intercepteur réplique peut être dans l'un des quatre états suivants : l'état *Réplique*, l'état *Intermédiaire*, l'état *Neutre* ou l'état *Primaire*.

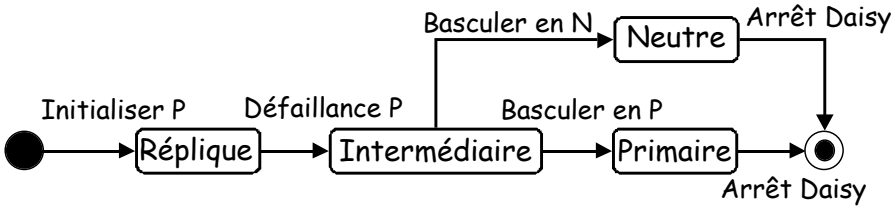


Figure 45. Les différents états de l'intercepteur *backup*

Lorsque l'intercepteur secondaire se trouve dans l'état *Réplique*, il assure la gestion de la mise à jour de l'état du serveur réplique. Si le serveur primaire défaille, par arrêt, l'intercepteur *PIS_PBRb* se met dans l'état intermédiaire *Intermédiaire*. Cet état sert à chercher un point de reprise. Lorsque le serveur réplique est mis à jour, l'intercepteur *PIS_PBRb* se met à l'état *Primaire* ou à l'état *Neutre*. Le premier état est choisi lorsque le nombre de répliques encore opérationnelles permet d'assurer l'exécution des mécanismes de réplication passive. En revanche, dans le cas contraire, c'est-à-dire quand l'intercepteur est associé au dernier serveur opérationnel, l'intercepteur réplique se met à l'état *Neutre* et n'assure aucun traitement.

Lors de l'initialisation de la plate-forme DAISY, le *PIS_PBRb* se met dans l'état *Réplique*. À l'inverse des deux premiers intercepteurs (i.e. intercepteur client et intercepteur primaire), l'intercepteur réplique ne récupère aucun *IOR* lors de son initialisation car nous avons choisi d'initialiser les serveurs secondaires avant le serveur primaire. Par conséquent, le *PIS_PBRb* attend l'interception de la première invocation pour récupérer aussi bien l'identificateur *IOR* du serveur auquel il est lui-même associé (i.e. serveur réplique), que l'identificateur du serveur primaire.

IV.1.4.2. Comportement dans l'état réplique

Comme le montre la Figure 46, les deux mécanismes fournis dans l'état réplique de l'intercepteur *PIS_PBRb* sont la gestion de la mise à jour de l'état du serveur réplique et la détection de la défaillance du serveur primaire. Ces mécanismes sont activés lors de l'interception des requêtes en provenance de l'intercepteur primaire, de l'intercepteur réplique lui-même ou d'un client.

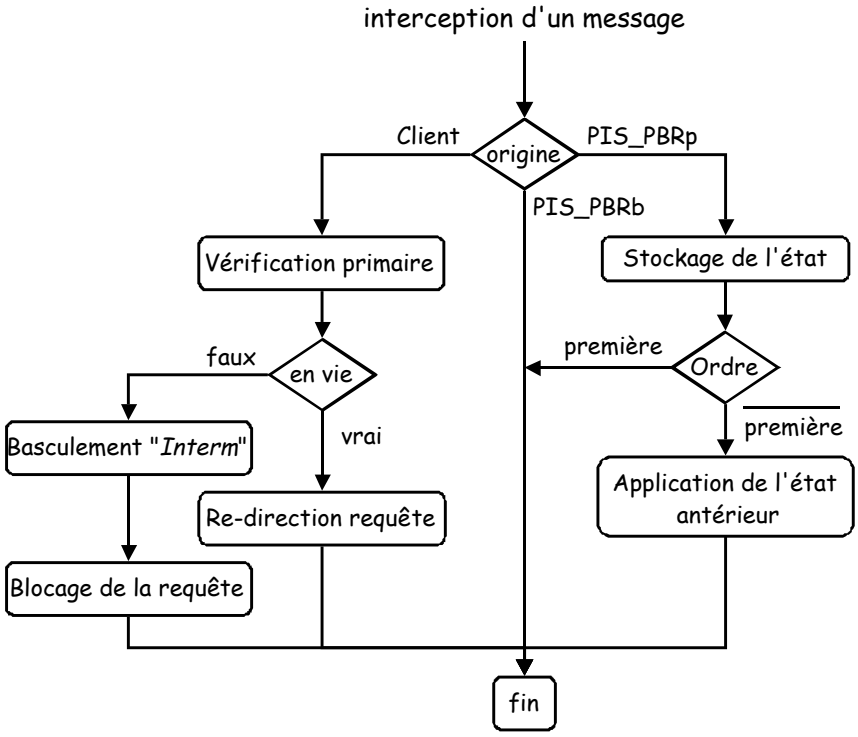


Figure 46. Les mécanismes associés à l'état *réplique* du *PIS_PBRb*

Lorsque l'intercepteur primaire envoie une requête de mise à jour de l'état au serveur secondaire, le *PIS_PBRb* intercepte cette requête puis l'identifie à partir du champ *method* du paramètre `requestinfo` de la méthode `receive_request()`, dans le cas présent cet identificateur représente `set_state()`. L'identification de cette requête enclenche le protocole de gestion de la mise à jour de l'état du serveur secondaire par le *PIS_PBRb*.

Soit E_i l'état du serveur principal obtenu suite au traitement de la ième requête. Lorsque l'intercepteur *PIS_PBRp* envoie la requête `set_state(`E_i`)` de mise à jour de l'état du serveur réplique, *PIS_PBRb* l'intercepte et lui applique le protocole, en quatre étapes, suivant:

i. Enregistrement de l'identificateur de la requête qui a généré l'état E_i.

ii. Enregistrement de l'état E_i au niveau de l'intercepteur.

iii. Mise à jour du serveur réplique avec le point de reprise précédent: invocation de la méthode `set_state(`E_{i-1}`)`.

iv. retourne une exception à l'intercepteur *PIS_PBRp*.

Dans le cas où le *PIS_PBRb* intercepte une requête de mise à jour pour la première fois, il n'invoquera pas la mise à jour de l'état du serveur secondaire.

P5 : Problème et incidence sur l'implémentation

L'intercepteur est obligé de transmettre la requête interceptée. C'est pour cette raison que dans notre implémentation, on doit générer une exception pour bloquer la propagation de la requête.

À la troisième étape du protocole de mise à jour de l'état, l'intercepteur réplique émet une requête à destination du serveur secondaire. Cette requête, qui est interceptée par le *PIS_PBRb*, est identifiée afin de ne pas enclencher le protocole de gestion de la mise à jour de l'état et provoquer aussi une boucle infinie (cf. problème 4 page 6). Nous associons des identificateurs de messages aux requêtes émises par l'intercepteur primaire afin de les distinguer de celles qui sont générées par l'intercepteur réplique.

En absence de fautes, le serveur secondaire ne reçoit que les requêtes de mise à jour. Ainsi, l'interception d'une requête dont le nom est différent de *set_state* est synonyme d'une déviation du service. La réception d'une telle requête n'est pas nécessairement due à la défaillance du serveur primaire. Il peut s'agir d'un problème temporaire de communication.

Avant de déclarer le serveur primaire défaillant, l'intercepteur secondaire vérifie sa disponibilité en invoquant la méthode *are_you_alive()*. Si le serveur primaire retourne une réponse, alors il s'agit d'une faute transitoire, de communication par exemple. L'intercepteur secondaire re-dirige la requête vers le serveur primaire qui est en effet non-défaillant. En revanche, si le serveur primaire ne répond pas, l'intercepteur secondaire s'engage dans une phase le conduisant au basculement dans l'état primaire.

P6 : Problème et incidence sur l'implémentation

L'intercepteur ne possède pas son propre brin d'exécution. C'est pour cette raison que dans notre implémentation, on ne peut pas mettre en œuvre une politique de détection de défaillance périodique.

IV.1.4.3. Comportement dans l'état intermédiaire

Lorsque l'intercepteur réplique est dans l'état intermédiaire, il peut intercepter deux types de requêtes : des requêtes de mise à jour de l'état et des requêtes en provenance des différents clients. Pour le premier type de requêtes, le *PIS_PBRb* applique le protocole de mise à jour de l'état. En revanche, pour le second type de requêtes, il recherche un état de reprise valide qui correspond à deux situations possibles :

- Le primaire a traité la requête et le secondaire a reçu le point de reprise ; ceci implique que le client n'a pas reçu la réponse.

- Le primaire n'a pas traité la requête ou a défailli par arrêt inopiné avant de transmettre le point de reprise au secondaire.

Le protocole de la gestion de la mise à jour de l'état du serveur répliqué est activé lors de l'interception d'une requête en provenance de l'intercepteur primaire (voir Figure 47). Cette situation se produit lorsque la requête redirigée, par l'intercepteur client, arrive au serveur réplique avant la requête de mise à jour, envoyée par le *PIS_PBRp*. Le protocole utilisé est celui que nous avons développé lorsque le *PIS_PBRb* se trouve dans l'état réplique.

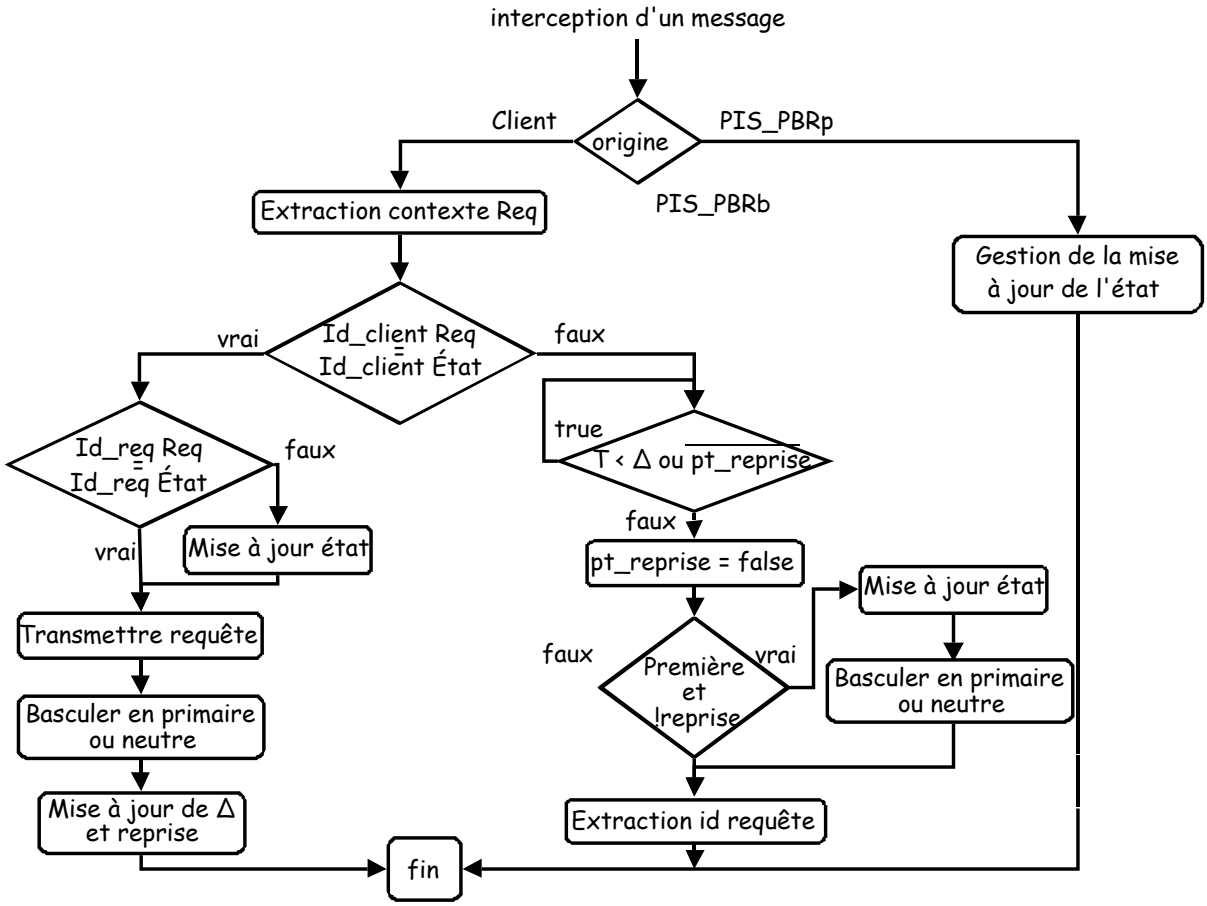


Figure 47. Les mécanismes associés à l'état *Interm* du *PIS_PBRb*

Comme le montre la Figure 47, lorsque le *PIS_PBRb* intercepte une requête en provenance d'un client, il commence par en extraire son contexte. Ensuite, il compare l'identificateur du client de la requête interceptée à celui de la requête qui a généré l'état stocké. Dans le cas où les identificateurs seraient identiques, l'intercepteur décidera d'appliquer ou non l'état stocké. En revanche, si les deux identificateurs sont différents, l'intercepteur bloque la requête en cours.

Dans le cas où l'identificateur du client et de l'état stocké sont identiques, l'intercepteur se charge de la recherche d'un point de reprise. Ainsi, il compare l'identificateur de la requête interceptée à celui de la requête qui a généré l'état stocké :

- Si les deux identificateurs de requêtes sont identiques, c'est-à-dire la défaillance est apparue après l'arrivée de la mise à jour de l'état au serveur réplique et avant le retour de la réponse, l'intercepteur ne met pas à jour l'état du serveur réplique. Le serveur traitera la requête en mode primaire afin de répondre au client. On peut noter qu'il

aurait été possible de réutiliser le résultat stocké avec la requête et l'état issus du primaire mais, à cause de la limitation des intercepteurs, on ne peut créer une réponse à une requête à partir d'un intercepteur.

P7 : Problème et incidence sur l'implémentation

L'intercepteur ne peut pas générer une réponse. C'est pour cette raison que dans notre implémentation, on doit traiter la requête de nouveau.

- Si les deux identificateurs de requêtes sont différents, c'est-à-dire la défaillance est apparue avant que la mise à jour ne soit parvenue au serveur réplique, l'intercepteur secondaire met à jour l'état du serveur réplique à partir du point de reprise correspondant à la requête précédente.

Suite à la décision d'appliquer ou non l'état stocké, l'intercepteur secondaire transmet la requête au serveur réplique, puis bascule dans le mode *Primaire* ou *Neutre*. Juste avant le basculement, le *PIS_PBRb* déclare le serveur répliqué en tant que primaire auprès du service de désignation, pour que tout nouveau client puisse lui envoyer des requêtes. Enfin, l'intercepteur met à jour la variable de temporisation « Δ » et la variable « reprise ». La première permet de borner temporellement l'attente des requêtes bloquées et la seconde permet d'inhiber la mise à jour de l'état du serveur réplique.

Dans le cas où l'identificateur du client et de l'état stocké sont différents, le *PIS_PBRb* bloque la requête interceptée. Pour que la requête ne soit pas bloquée indéfiniment, il évalue l'expression suivante :

$$(T<\Delta) \text{ ou } (!pt\_reprise) \qquad (1)$$

Le premier terme de cette expression ($T < \Delta$) correspond à une temporisation qui bloque les requêtes. En effet, nous bloquons toutes les requêtes qui proviennent d'un client différent de celui qui a engendré le dernier état stocké pour s'assurer que ce dernier a bien reçu une réponse (i.e. cas d'occurrence d'une défaillance du serveur après l'envoi de la mise à jour aux répliques et avant l'arrivée de la réponse au client). Le second terme *pt_reprise* permet de délivrer les requêtes d'une manière séquentielle. En effet, dès qu'une requête est débloquée, cette variable est mise à *faux*, pour en libérer qu'une et une seule.

Lorsque le délai d'attente est écoulé, c'est-à-dire que l'on considère que le client émetteur de la dernière requête avant la défaillance du serveur primaire a reçu sa réponse, la première requête qui arrive à obtenir le *mutex* de *pt_reprise* déclenche la mise à jour d'un état de poursuite. L'intercepteur secondaire met à jour le contenu de la variable *pt_reprise* pour bloquer les autres requêtes. Ensuite, il s'assure que le système n'a pas trouvé un point de reprise pour mettre à jour l'état du serveur réplique, puis il bascule dans le mode *Primaire* ou *Neutre*. Enfin, il extrait l'identificateur de la requête pour qu'il soit utilisé, lorsque l'intercepteur se met à l'état *Primaire*, dans l'élaboration de la requête de mise à jour de l'état.

Lors du développement de la plate-forme DAISY, nous avons relevé que la mise en attente d'une requête ne bloque pas l'intercepteur *PIS_PBRb*, puisque chaque requête possède son propre brin d'exécution. Ainsi il a fallu que nous prêtions une attention particulière aux variables partagées pour les protéger des accès concurrents, à savoir les variables contenant les identificateurs des requêtes.

IV.1.4.4. Comportement dans l'état Neutre

Dans l'état *Neutre*, l'intercepteur réplique *PIS_PBRb* intercepte toutes les requêtes qui arrivent au niveau de l'interface du serveur réplique. Aucun traitement relatif aux mécanismes de réplication passive n'est réalisé. En effet, lorsque l'intercepteur se met dans cet état, cela veut dire que le nombre de serveurs opérationnels n'est pas suffisant pour fournir un service répliqué.

IV.2. MISE EN ŒUVRE DE LA REPLICATION ACTIVE

Dans cette partie, nous présentons l'implémentation des mécanismes de réplication active à l'aide des intercepteurs CORBA. Comme pour le cas de la réplication passive, nous avons implémenté deux des composants de tolérance aux fautes (voir la section 2 du chapitre III), à savoir le composant *CC-C/S* et le composant *ACR*. Le premier qui concerne la partie du protocole côté client est identique au cas précédent. Nous réutilisons donc celui que nous avons développé pour la réplication passive. Le second composant assure la diffusion des messages en respectant un ordre total, de façon simple et sans avoir recours à des protocoles de diffusion atomique. Par ailleurs, comme l'objectif de la réplication active est de tolérer les défaillances par valeurs, ce composant fournit un algorithme de vote majoritaire.

IV.2.1. Initialisation du mode actif

À l'initialisation d'une application serveur, dans le cas de la réplication active, la plate-forme DAISY instancie plusieurs serveurs d'application et associe à chacun d'entre eux un intercepteur serveur de réplication active *PIS_ACR*. Un seul parmi l'ensemble de ces serveurs est initialisé en tant que serveur principal, les autres sont des serveurs subsidiaires. Cette différence ne réside pas au niveau du code de l'application, elle est liée au type de l'intercepteur serveur qui lui est associé. En effet, si la plate-forme associe l'intercepteur *PIS_ACRp* à l'application, le serveur sera principal. En revanche, si elle lui associe l'intercepteur *PIS_ACRs*, le serveur sera subsidiaire. Cette différence réside dans le fait que nous utilisons un protocole d'ordre asymétrique [Ezhilchelvan 1995] ; l'intercepteur principal est utilisé comme séquenceur pour ordonner les messages émis par les différents clients.

Comme pour le cas de la réplication passive, toute application cliente qui utilise la plate-forme DAISY est initialisée avec le composant *CC-C/S*, un intercepteur client *PIC*.

IV.2.2. Intercepteur Principal

L'intercepteur *PIS_ACRp* a pour but d'ordonner d'une manière séquentielle les requêtes émises par les différents clients, de diffuser d'une manière atomique ces requêtes aux serveurs du groupe de répliques (i.e. serveurs subsidiaires), de retourner des réponses correctes aux différents clients en leur appliquant un vote majoritaire. Le serveur principal et son intercepteur sont liés de telle sorte que l'arrêt inopiné de l'un implique celui de l'autre. L'intercepteur ne peut donc assurer que le recouvrement d'erreurs par valeur (dans le cas où le serveur principal ou l'un des serveurs subsidiaires retourneraient des réponses erronées) ainsi que la tolérance de l'arrêt inopiné des serveurs subsidiaires.

IV.2.2.1. Initialisation

Lors de l'initialisation de la plate-forme DAISY avec la réplication active, les différents serveurs d'application utilisent la méthode `subscribe`, héritée de la classe `FT_Serveur` (voir la section 3 du chapitre III), pour s'enregistrer en tant que principal ou subsidiaire auprès du service désignation de CORBA.

Dans le cas où le serveur s'enregistre comme principal, nous lui associons un intercepteur principal *PIS_ACRp*. Ce dernier récupère les *IOR* des différents serveurs lancés avec le profil subsidiaire pour qu'il puisse leur diffuser les requêtes qu'il a interceptées. Une fois que le *PIS_ACRp* et le serveur qui lui est associé sont bien initialisés, ils se mettent en attente des requêtes des clients.

Comme l'identificateur d'un serveur est généré après l'initialisation de l'intercepteur qui lui est associé, l'intercepteur principal récupère l'*IOR* du serveur principal lors de l'interception de la première invocation (cf. problème 1 page 4). Cet identificateur est utilisé ultérieurement pour récupérer l'état du serveur principal.

IV.2.2.2. Comportement

Les différents traitements dont nous allons présenter le déroulement, correspondent aux quatre mécanismes suivants :

- l'ordonnancement séquentiel des requêtes en provenance des clients,
- la diffusion des messages,
- le vote majoritaire, et
- le recouvrement des erreurs par valeur.

Comme l'intercepteur ne possède pas son propre brin d'exécution (cf. problème 6 page 9), l'activation des mécanismes qui implémentent les différents traitements est commandée non seulement par l'interception des requêtes en provenance des clients, des intercepteurs subsidiaires et de l'intercepteur principal lui-même, mais aussi des réponses à ces requêtes.

A- Ordonnancement séquentiel des requêtes

L'ordonnancement séquentiel des requêtes est déterminant pour garantir la cohérence des traitements effectués par les serveurs subsidiaires. En effet, en ne laissant passer qu'une requête à la fois, l'intercepteur principal assure que le serveur auquel il est associé et les différents serveurs subsidiaires traitent la même requête au même moment logique.

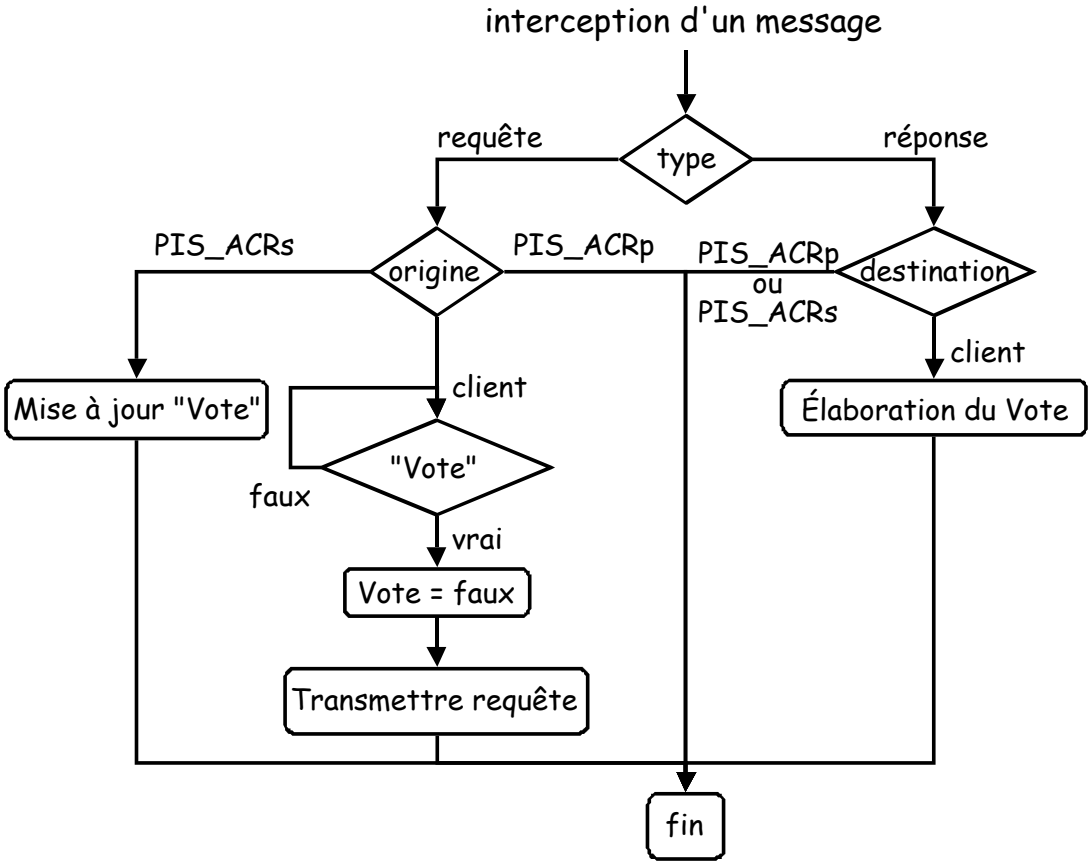


Figure 48. Protocole d'ordonnancement des requêtes

Comme le montre la Figure 48, ce mécanisme est activé à chaque fois qu'une requête en provenance d'un client est interceptée par le *PIS_ACRp*. Toute requête mise en attente par l'intercepteur principal n'est libérée que lorsque ce dernier trouve une réponse correcte lors du vote majoritaire. Dans l'implémentation, nous utilisons une variable particulière (la variable d'état *Vote*) permettant de modéliser et de contrôler l'état de l'intercepteur : état d'élaboration d'une réponse majoritaire et état débloquant. Lorsque cette variable possède la valeur *faux*, elle indique le blocage des requêtes interceptées. Pour assurer qu'une seule requête soit débloquée, lorsque la variable d'état *Vote* passe à la valeur *vrai*, l'accès en lecture à cette variable d'état est protégé par un sémaphore. Toute requête libérée sera transmise par la suite au serveur principal pour être traitée.

L'ordonnancement ne s'applique qu'aux requêtes en provenance des clients. Les requêtes interceptées par le *PIS_ACRp* et qui sont relatives aux mécanismes non-fonctionnels ne sont pas assujetties à l'ordonnancement pour ne pas bloquer l'application. En effet, les requêtes en

provenance des intercepteurs subsidiaires et de l'intercepteur principal lui-même, sont générées par le mécanisme de recouvrement d'erreurs que nous allons traiter par la suite. Elles sont identifiées comme provenant d'un intercepteur par leur nom `set_state` et leur contexte.

Par ailleurs, seules les réponses à destination des clients interviennent dans l'évolution du protocole d'ordonnancement des requêtes. Les réponses à destination de l'intercepteur principal lui-même et de l'intercepteur subsidiaire font partie des interactions relatives au recouvrement d'erreur ; ainsi, elles ne sont soumises à aucun post-traitement.

B- Diffusion des messages

Le mécanisme de diffusion est déclenché suite au passage d'une requête par le protocole d'ordonnancement. En effet, juste avant de transmettre la requête au serveur principal, le *PIS_ACRp* diffuse cette requête aux différents serveurs subsidiaires. Pour diffuser une requête, l'intercepteur principal commence par créer dynamiquement une nouvelle requête en utilisant l'interface des invocations dynamiques (*DII*, voir la section 1 du chapitre I). Ensuite, il met à jour ses attributs, à savoir le nom de la méthode et les paramètres, à partir des informations qu'il récupère de la requête interceptée. Enfin, il invoque cette nouvelle requête sur chaque serveur subsidiaire. Toutes les réponses à cette requête seront stockées pour fournir une entrée de l'algorithme de vote. Dans le cas où un serveur subsidiaire ne fournit pas de réponse, suite à une défaillance par arrêt, l'intercepteur principal notifie le gestionnaire de réplication de cette défaillance afin de lancer un nouveau serveur subsidiaire.

Pour créer une requête dynamique, nous utilisons la méthode `create_request` qui possède 4 paramètres : le contexte de la requête, le nom de l'opération, la liste des arguments et le résultat. Ces quatre paramètres possèdent les types suivant : `Context`, `String`, `NVList` et `NamedValue`. Ainsi, pour créer une requête, l'intercepteur doit construire chaque paramètre de la méthode `create_request`. Le contexte est récupéré en invoquant la méthode `get_default_context` de l'interface de l'ORB. Le nom de l'opération est récupéré à partir du paramètre de la méthode activée par l'intercepteur, en l'occurrence le paramètre `request_info` de la méthode `receive_request`. La liste des arguments fait appel à la méthode `create_list` de l'interface de l'ORB. Pour créer le paramètre du résultat, nous utilisons aussi la méthode `create_named_value` de l'interface de l'ORB.

Comme l'intercepteur standard n'a aucun moyen de récupérer l'identificateur de l'interface de l'ORB, il ne peut pas créer de requêtes dynamiques. Pour résoudre ce problème, nous utilisons le contexte de la première requête pour passer la référence de l'ORB à l'intercepteur pour qu'il puisse utiliser les méthodes qui y sont définies.

P8 : Problème et incidence sur l'implémentation

L'intercepteur ne peut pas utiliser l'interface des invocations dynamiques. C'est pour cette raison que dans notre implémentation, on doit passer la référence de cette interface dans le contexte de la première invocation.

C- Vote majoritaire

Le protocole de vote majoritaire est simple, il est activé lors de l'interception d'une réponse à destination d'un client. À ce moment, l'intercepteur principal compare le résultat retourné par la requête à ceux qui sont retournés par les différents serveurs subsidiaires. Trois cas se présentent :

- Toutes les réponses sont identiques : l'intercepteur met à jour la variable *Vote* pour débloquer une des requêtes mises en attente, ensuite il retourne la réponse au client.

- L'intercepteur aboutit à un vote majoritaire, alors qu'il existe des réponses erronées. Nous distinguons deux sous-cas :

    - la réponse du serveur principal est correcte : l'intercepteur commence par mettre à jour la variable *Vote* pour débloquer une des requêtes mises en attente, ensuite il déclenche le mécanisme de recouvrement d'erreur pour les serveurs subsidiaires défaillants, et enfin retourne la réponse au client.

    - la réponse du serveur principal est erronée : l'intercepteur déclenche le mécanisme de recouvrement et la réponse ne sera pas délivrée directement au client. En effet, les intercepteurs ne peuvent pas modifier le contenu d'une requête entrante ni celui d'une réponse. Ainsi, l'intercepteur principal ne peut changer le résultat avant de le transmettre au client.

- L'intercepteur ne trouve pas une réponse majoritaire : l'intercepteur transmet la réponse élaborée par le serveur principal, ensuite il active le mécanisme de recouvrement d'erreur. Ce choix peut correspondre à une réponse erronée, il y a en effet eu simultanément un nombre de fautes trop important, mais il représente le comportement du système sans réplication.

Il existe deux variantes de réplication active : avec vote à la source (*validate before propagate*) ou à la destination (*propagate before validate*). Si notre plate-forme fournit la première variante de la réplication active, nous ne pouvons pas implémenter la seconde variante avec la version actuelle des intercepteurs. En effet, les réponses relatives à la diffusion d'un message par notre plate-forme sont retournées à l'intercepteur principal. Pour que le vote soit réalisé à la destination, les réponses doivent être retournées directement au client. L'intercepteur principal, responsable de la diffusion d'un message, ne peut utiliser que l'exception *forward_request* pour que la réponse à ce message soit retournée directement au client. Néanmoins, cette exception ne peut rediriger la requête qu'à un seul serveur. Ainsi, nous ne pouvons pas avoir plusieurs réponses retournées au client.

P9 : Problème et incidence sur l'implémentation

L'intercepteur ne peut pas rediriger la requête interceptée à plusieurs serveurs. C'est pour cette raison que dans notre implémentation, on applique un vote à la source et non pas à la destination.

D- Recouvrement d'erreurs par valeur

Le mécanisme de recouvrement d'erreur est déclenché suite à la détection par le mécanisme de vote majoritaire d'une défaillance en valeur d'un serveur au sein du groupe de répliques. Ce mécanisme se comporte d'une manière différente selon que le serveur défaillant est subsidiaire ou principal.

En effet, dans le premier cas, l'intercepteur principal génère un point de reprise en récupérant l'état du serveur principal. Ensuite, il envoie à chaque serveur subsidiaire une requête de mise à jour de l'état.

En revanche, dans le cas où le serveur principal est défaillant, le *PIS_ACRp* redirige la requête vers un serveur subsidiaire correct. L'intercepteur associé à ce serveur subsidiaire, que nous présenterons dans la section suivante, se charge de la terminaison du mécanisme de recouvrement.

La re-direction de la requête, dans le second cas, est inévitable bien que le *PIS_ACRp* détienne la réponse correcte. En effet, les intercepteurs ne peuvent pas modifier le contenu d'une requête entrante ni celui d'une réponse. Ainsi, il ne peut pas changer le résultat avant de le transmettre au client. C'est pour cette raison que nous avons opté pour le renvoi de la requête vers un serveur correct afin d'être traitée.

P10 : Problème et incidence sur l'implémentation

L'intercepteur ne peut pas changer le contenu d'une réponse. C'est pour cette raison que dans notre implémentation, on redirige la requête à un serveur subsidiaire correct.

IV.2.3. Intercepteur Subsidiaire

L'intercepteur serveur subsidiaire *PIS_ACRs* se charge de la prise de points de reprises, rendue nécessaire par une limitation des intercepteurs, et d'assurer une partie du mécanisme de recouvrement d'erreurs.

IV.2.3.1. Modélisation et Initialisation

Comme le montre la Figure 49, l'intercepteur subsidiaire peut-être dans l'un des trois états suivants : l'état *Subsidiaire*, l'état *Neutre* ou l'état *Principal*.

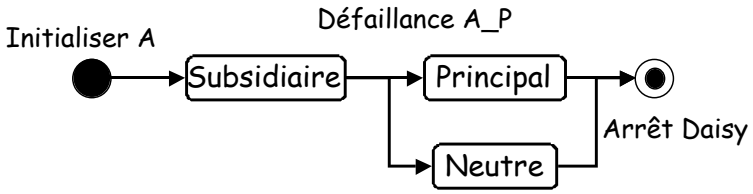


Figure 49. les différents état de l'intercepteur subsidiaire

Lorsque l'intercepteur subsidiaire se trouve dans l'état *Subsidiaire*, il assure la prise de points de reprises,, et une partie du mécanisme de recouvrement d'erreurs. Si le serveur principal défaille par arrêt (i.e. *Défaillance A_P*), l'intercepteur *PIS_PBRs* bascule dans l'état *Principal* ou *Neutre*. Le choix entre l'état *Principal* et l'état *Neutre* est régi par le nombre de répliques disponibles. En effet, si ce nombre ne permet pas la mise en œuvre de la réplication active, l'intercepteur subsidiaire bascule à l'état *Neutre*. En revanche, si ce nombre est suffisant, le *PIS_ACRs* bascule à l'état *Principal*.

A l'initialisation (i.e. *Initialiser A*), l'intercepteur *PIS_ACRs* se met dans l'état *Subsidiaire*. Il ne récupère l'identificateur *IOR* du serveur auquel il est associé ainsi que celui du serveur principal que lors de l'interception de la première invocation à destination du serveur subsidiaire.

Nous pouvons noter une grande différence entre le diagramme d'états de l'intercepteur réplique (i.e. cas de la réplication passive) avec celui de l'intercepteur subsidiaire (i.e. cas de la réplication active). Dans le cas de la réplication active, l'intercepteur subsidiaire ne possède pas d'état intermédiaire. En effet, dans la réplication active, nous n'avons pas besoin de chercher un point de reprise ou de poursuite pour mettre à jour l'état du serveur subsidiaire. Cet état est issu du traitement par le serveur subsidiaire des requêtes diffusées par l'intercepteur principal.

IV.2.3.2. Comportement

Les différents traitements dont nous allons présenter le déroulement, correspondent aux deux mécanismes suivants : la prise de points de reprise et le recouvrement d'erreur.

A- Prise de points de reprise

Ce mécanisme est activé lors de l'interception d'une réponse à destination de l'intercepteur principal. Ainsi, avant de retourner la réponse, l'intercepteur subsidiaire récupère l'état du serveur auquel il est associé en invoquant la méthode `get_state()`. Cet état est stocké et sera utilisé lors de l'exécution du mécanisme de recouvrement d'erreur.

Comme les intercepteurs CORBA ne permettent pas de générer une réponse (cf. problème 7 page 11), le serveur réplique doit la calculer de nouveau. Ainsi, l'intercepteur doit récupérer l'état de la réplique à chaque invocation pour que dans le cas où le serveur sera amené à re-générer une réponse, l'intercepteur lui applique l'état stocké afin de récupérer la même réponse.

Tout comme le cas pour l'intercepteur secondaire, l'intercepteur subsidiaire ne doit pas appliquer ce mécanisme aux réponses relatives à ses propres requêtes, par exemple celles qu'il génère lors de la prise de point de reprise. Si tel n'était pas le cas, l'intercepteur se bloquerait à cause du traitement récursif des réponses interceptées.

Par ailleurs, le mécanisme de prise de points de reprises déclenche l'interception des réponses à destination du *PIS_ACRs*. Ces dernières, que nous identifions à partir de leurs noms, doivent être transmises à l'intercepteur sans activer le mécanisme de génération de points de reprise afin d'éviter la récursivité de son activation.

B- Recouvrement d'erreurs

Il existe deux types d'erreurs supportées par le mécanisme de réplication active : la défaillance par arrêt et la défaillance en valeur d'un serveur. L'intercepteur subsidiaire intervient dans le recouvrement :

- lors de la défaillance par arrêt du serveur principal, et
- lors de la défaillance en valeur du serveur principal ou du serveur subsidiaire auquel il est associé.

Dans le cas de la défaillance par arrêt du serveur principal, le mécanisme de recouvrement d'erreurs est activé lors de l'interception d'une requête en provenance de l'intercepteur client *PIC*, comme c'était le cas pour la réplication passive. Cette requête est identifiée par l'intercepteur principal à partir du contexte que l'intercepteur client lui associe. Lorsque cette requête est identifiée, l'intercepteur subsidiaire commence par vérifier l'état du serveur principal en invoquant la méthode *ping*. Ensuite, si le serveur principal répond, le *PIS_ACRs* lui redirige la requête, en supposant que la défaillance était transitoire. En revanche, si le serveur principal ne répond pas, l'intercepteur subsidiaire récupère le nombre de répliques encore opérationnelles auprès du gestionnaire de réplication. Enfin, si leur nombre permet encore de munir l'application de la réplication active, le *PIS_ACRs* se transforme en un *PIS_ACRp* en inscrivant le serveur qui lui est associé comme serveur principal et demande la référence des autres serveurs subsidiaires auprès du service de désignation. Par contre, si le nombre de serveurs ne permet pas l'aboutissement des mécanismes de réplication active, l'intercepteur subsidiaire bascule dans l'état Neutre.

Dans le cas de la défaillance en valeur du serveur principal, l'intercepteur *PIS_ACRp* redirige la requête qui a provoqué cette défaillance au serveur subsidiaire. L'interception de cette requête par le *PIS_ACRs* active le mécanisme de recouvrement d'erreurs. Cependant, en mode de fonctionnement en absence de fautes, l'intercepteur subsidiaire intercepte aussi les requêtes diffusées par l'intercepteur principal. Pour pouvoir distinguer les requêtes redirigées des requêtes diffusées, l'intercepteur subsidiaire analyse le contexte de chaque requête interceptée. Dans le cas où cette dernière serait redirigée, l'intercepteur subsidiaire applique l'état qu'il a stocké précédemment (point de reprise) au serveur qui lui est associé. Puis, il lui transmet la

requête redirigée. Enfin, et au retour de la réponse, il met à jour l'état du serveur principal en invoquant la méthode `set_state()` avec l'état stocké.

Nous tenons à souligner que lors du recouvrement de ce type de défaillances, l'intercepteur subsidiaire doit distinguer les réponses relatives à la diffusion d'un message par l'intercepteur principal de celles relatives à la sa re-direction. En effet, si le premier type enclenche le mécanisme de prise de point de reprise, le second type n'a pas besoin de récupérer l'état du serveur subsidiaire car celui-ci est déjà récupéré car la requête est rejouée (cf. problème7 page 11).

Dans le cas de la défaillance en valeur du serveur subsidiaire, le mécanisme de recouvrement d'erreurs est activé lors de l'interception de la requête `set_state` en provenance de l'intercepteur principal. Une fois cette requête identifiée, l'intercepteur subsidiaire commence par récupérer l'information qu'elle contient, à savoir l'état du serveur principal, ensuite stocke cette information, et enfin transmet la requête au serveur qui lui est associé afin de mettre à jour son état.

IV.3. GESTION DE LA PLATE-FORME DAISY

Dans cette section, nous présentons les différents composants utilitaires de la plate-forme DAISY qui sont utilisés par les composants de tolérance aux fautes, à savoir l'usine à objets et le gestionnaire de réplication. Nous présentons, par ailleurs, l'apport des composants utilitaires pour le recouvrement des défaillances des serveurs d'applications.

IV.3.1. Usine à objets

L'usine à objets est un service qu'utilise le gestionnaire de réplication pour lancer un serveur d'application. Le lancement d'un nouveau serveur est demandé suite à la défaillance par arrêt d'un serveur appartenant au groupe de répliques. Comme la plate-forme DAISY supporte deux types de réplications (i.e. réplication passive et active), l'usine à objets permet de lancer différents types de serveurs. En effet, dans le cas où l'application est munie de la réplication passive, l'usine à objets permet de créer des serveurs primaires et des serveurs répliques. En revanche, dans le cas où l'application serait munie de la réplication active, l'usine à objets crée des serveurs principaux et des serveurs subsidiaires.

Il existe deux solutions pour mettre en œuvre le service usine à objets, la première basée sur une approche langage, la deuxième sur une approche service.

Avec la première approche, l'usine à objets effectue l'instanciation de chaque nouveau serveur comme étant un objet java, en utilisant la méthode `new`. En effet, chaque fois que l'usine veut créer un nouveau serveur, elle crée une nouvelle instance de la classe `serveur_impl`, qui implémente l'interface *idl* du serveur d'application.

Avec la deuxième approche, l'usine à objets lance chaque nouveau serveur comme étant un objet CORBA, en utilisant un lanceur d'application. Ce dernier est responsable de la création d'une instance de la classe `serveur_impl`. Par ailleurs, le lanceur d'application est activé au niveau de l'usine à objets comme étant un programme java, c'est-à-dire, en utilisant la commande `java.lang.exec()`.

L'utilisation de la première approche, relative à l'aspect langage, lie la plate-forme DAISY aux applications qui lui sont associées et la rend dépendante des serveurs auxquels elle est associée. En effet, si nous modifions le nom de la classe qui implémente l'interface *idl* du service, nous serons amené à changer le code de l'usine à objet et de la recompiler. En revanche, l'utilisation d'un lanceur, permet de rendre l'usine à objets indépendante de l'application, puisqu'il suffit de passer le nom du lanceur à l'usine pour pouvoir y exécuter la ligne de commande qui l'initialise. Comme notre objectif est de fournir une plate-forme générique et indépendante des applications qui l'utilisent, nous avons opté pour l'utilisation de la seconde approche.

Comme le montre la Figure 50, en choisissant la seconde approche, nous fournissons à l'usine à objets l'emplacement et le nom du lanceur de l'application `n_lanceur` ainsi que le nom de la machine sur laquelle nous voulons exécuter le serveur d'application `m_cible`. L'usine à objets nous retourne alors la référence de l'objet qu'elle a créé, `r_objet`. Comme à chaque lanceur correspond un seul serveur `Serveur_impl`, si nous voulons lancer plusieurs serveurs d'application, nous devons créer plusieurs lanceurs. Par ailleurs, la classe `Lanceur` doit hériter de la classe `Thread` puis implémenter la méthode `run` pour que chaque serveur lancé soit exécuté dans un brin d'exécution différent des autres.

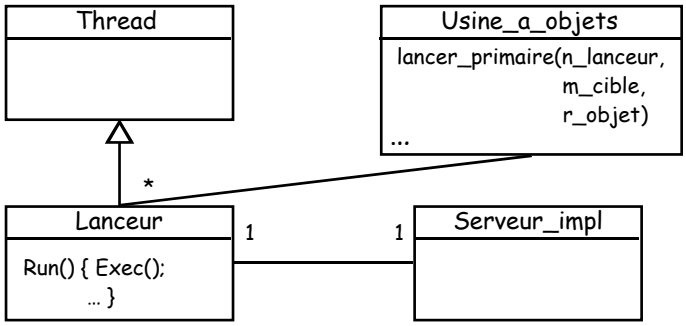


Figure 50. Diagramme de classe de l'usine à objets

IV.3.2. Gestionnaire de réplication

Le gestionnaire de réplication est un service CORBA permettant la saisie des consignes de l'administrateur de la plate-forme et la prise en compte des notifications émises par les composants de tolérance aux fautes. Selon les informations qu'il récupère, il utilise les données fournies par l'administrateur pour initialiser l'ensemble des composants d'une application répliquée. Il analyse aussi les données fournies par les composants de tolérance aux fautes pour gérer le bon fonctionnement de la plate-forme, qui doit répondre aux consignes initiales, en tenant compte de l'évolution du système global.

Pour répondre à ces deux besoins différents, le gestionnaire de réplication se comporte en tant que client par rapport à l'administrateur de la plate-forme et en tant que serveur vis-à-vis des différents composants de tolérance aux fautes.

Comme le montre la Figure 51, la classe console, qui représente la partie client, permet à l'administrateur d'initialiser la plate-forme. Le notificateur offre une interface qui permet aux intercepteurs (i.e. composants de tolérance aux fautes) d'informer le gestionnaire de réplication de la défaillance d'une réplique. Nous invoquons la méthode `cree_n_replique` pour créer une nouvelle réplique dans le cas de la réplication passive, et la méthode `cree_n_subsidiaire` dans le cas de la réplication active.

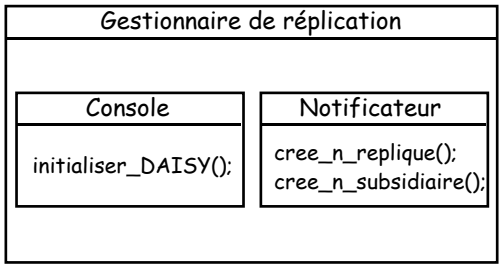


Figure 51. Composants du gestionnaire de réplication

IV.3.2.1. Initialisation de la plate-forme

Pour initialiser la plate-forme DAISY, l'administrateur doit fournir deux types d'informations : des informations relatives à l'application à répliquer et des informations concernant la technique de réplication que la plate-forme doit fournir.

Nous avons indiqué dans la section 3 que le lancement d'un serveur d'application est réalisé selon une approche service, c'est-à-dire en fournissant l'emplacement et le nom de la classe principale qui lance le service. Ainsi, lors de l'initialisation de la plate-forme DAISY l'administrateur est amené à fournir le nom de l'application et le chemin d'accès aux fichiers « .class » (i.e. la variable `classpath`).

L'administrateur est amené, par ailleurs, à préciser le mécanisme de réplication à fournir. Ensuite, il doit fournir le nombre de serveurs dont le groupe de répliques doit se composer. Enfin, il doit choisir l'indice de distribution de l'application répliquée. Ceci déterminera le nombre de machines à utiliser ou le nombre de serveurs par machine dans le cas où l'on dispose d'un nombre limité de machines.

Une fois ces informations saisies, le gestionnaire de réplication lance l'usine à objets puis lui demande la création des différents serveurs avec leurs intercepteurs associés. Pour cela, il fournit à l'usine à objets le nom de l'application, le `classpath` et le nom de la machine sur laquelle l'application s'exécutera.

IV.3.2.2. Gestion des défaillances

Lors de l'occurrence d'une défaillance par arrêt d'un serveur d'application, cette anomalie est détectée, selon le serveur, par les intercepteurs *PIS_ACRp* ou *PIS_ACRs* dans le cas de la réplication active, et par les intercepteurs *PIS_PBRp* et *PIS_PBRb* dans le cas de la réplication passive. Ces différents composants notifient le gestionnaire de réplication de cette défaillance d'un serveur pour qu'il puisse la pallier.

Les paramètres de tolérance aux fautes, comme le nombre de répliques ou le mode de réplication, sont centralisés au niveau du gestionnaire de réplication et non pas au niveau des différents intercepteurs; cette solution à pour but d'alléger leur l'état et les traitements dont ils sont responsables.

En effet, l'état d'une application renferme aussi bien les attributs relatifs à son aspect fonctionnel que ceux qui sont relatifs à son aspect non-fonctionnel. Dans les applications que nous considérons, ces aspects sont localisés dans des objets distincts (i.e. application et intercepteur). Si l'état du premier type d'objets est récupérable d'une manière standard en utilisant la sérialisation JAVA lors de la prise des points de reprises, l'état des intercepteurs CORBA ne peut pas l'être. Par conséquent, nous devons minimiser leur état, voire même l'éviter, en utilisant d'une part le gestionnaire de réplication comme support, et d'autre part le contexte des requêtes pour que l'information relative à la tolérance aux fautes soit partagée par l'ensemble des intercepteurs.

Par ailleurs, les intercepteurs étant déjà le centre de plusieurs traitements complexes, nous n'avons pas ajouté les traitements relatifs à la mise en service de nouvelle réplique dans le cas de l'arrêt d'un serveur du groupe. Ce choix a pour but d'améliorer les performances de notre plate-forme.

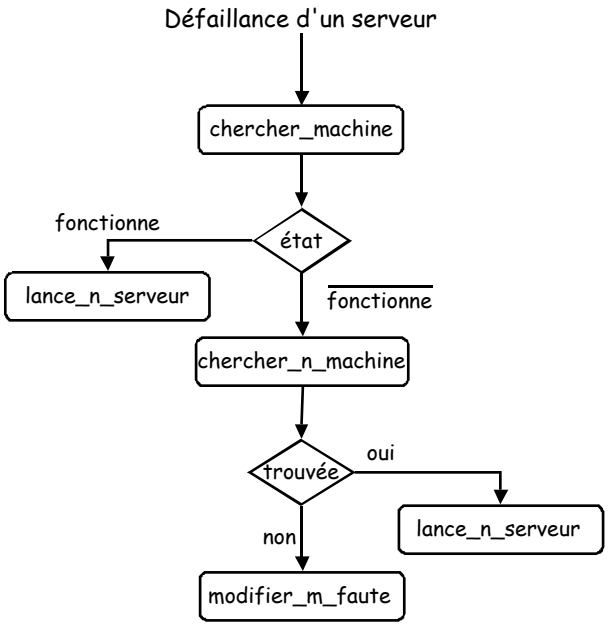


Figure 52. Traitement de l'arrêt d'un serveur

Comme le montre la Figure 52, lorsque le gestionnaire de réplication est notifié de la défaillance d'un serveur du groupe de réplication, pour n'importe quelle mode de réplication, il commence par vérifier si la machine qui héberge le serveur en question est opérationnelle (*état*). Si la machine fonctionne, le gestionnaire de réplication lance le nouveau serveur : il lance un serveur subsidiaire dans le cas de la réplication active et un serveur réplique dans le cas de la réplication passive. Dans le cas où la machine ne fonctionne pas, le gestionnaire de réplication se met à la recherche d'une autre machine pour y lancer le nouveau serveur. S'il n'en trouve pas une, il n'acquitte pas l'intercepteur demandeur, et selon le nombre de répliques opérationnelles, la plate-forme remplira le contrat de réplication ou délivrera un service de tolérance aux fautes dégradé.

Le service dégradé de tolérance aux fautes concerne le nombre de fautes à tolérer qui est relatif au nombre de serveurs opérationnels; en effet, dans le cas où le gestionnaire de réplication n'arrive pas à lancer un nouveau serveur, c'est le nombre de fautes tolérées qui est revu à la baisse.

IV.4. ÉVALUATION DES INTERCEPTEURS CORBA ET PROPOSITION DE SOLUTIONS D'UN POINT DE VUE DU DEVELOPPEMENT

Nous avons décelé lors de la phase d'implémentation des mécanismes de réplication des inconvénients qui ont influencé leur mise en œuvre. Nous allons, dans ce qui suit, discuter des limites que nous avons identifiées tout en proposant des solutions.

IV.4.1. Association intercepteur/serveur non identifiable

L'intercepteur serveur n'a aucun moyen simple pour récupérer l'identificateur du serveur auquel il est associé. Les paramètres avec lesquels il est initialisé et les informations associées aux requêtes qu'il intercepte ne renferment pas l'*IOR* du serveur cible.

La norme CORBA définit l'objet `ORBInitializer` pour déclarer et enregistrer un ou plusieurs intercepteurs. Lors de son initialisation, l'ORB fait appel aux différents `ORBInitializer` pour initialiser les différents intercepteurs en invoquant leurs constructeurs respectifs. Pour que l'intercepteur puisse, à l'initialisation, récupérer l'IOR du serveur auquel il est associé, nous devons passer cet identificateur via son constructeur. Si l'interface offerte par l'objet `ORBInitializer` permet de récupérer la référence des services initiaux (par exemple le serveur de désignation), elle ne permet pas de récupérer l'IOR du serveur auquel l'intercepteur sera associé.

Lorsqu'un *PIS* intercepte une requête, il a accès aux données contenues dans le paramètre `ServerRequestInfo` de la méthode activée. Si ce paramètre contient la quasi-totalité des informations concernant une requête, il ne renferme pas l'identificateur du serveur cible. La

seule information relative au serveur cible est une séquence d'octets nous renseignant sur le nom de son interface *IDL*.

Pour que l'intercepteur puisse récupérer l'IOR du serveur auquel il est associé à l'initialisation de l'ORB, il faut que la création du serveur précède l'enregistrement de l'intercepteur. En revanche, pour que l'intercepteur puisse récupérer cet identificateur lors de l'interception d'une requête, il faut que le paramètre `ServerRequestInfo` puisse renfermer une telle information. Cette extension semble plus facile, puisque l'intercepteur client peut récupérer l'identificateur du serveur cible via le champ `target` du paramètre `ClientRequestInfo`.

En utilisant la version augmentée des intercepteurs que nous proposons, présentée à la section 5 du chapitre III, chaque intercepteur est considéré comme un objet CORBA. Si l'intégrateur de l'application initialise le serveur avant l'intercepteur, ce dernier pourra récupérer l'IOR du serveur auquel il est associé et mettre à jour l'attribut « Identificateur de l'objet associé ».

IV.4.2. Absence d'une politique d'ordonnancement

Les intercepteurs standards, serveur et client, n'implémentent pas une file d'attente pour les requêtes qui ont été interceptées.

En effet, nous ne pouvons pas définir une politique d'ordonnancement des requêtes par rapport à l'ordre de leur interception. Par exemple, si l'intercepteur intercepte une requête *R1* juste avant une requête *R2* et que le traitement non-fonctionnel de la première requête est plus long que la seconde, l'intercepteur délivre la requête *R2* au serveur avant la requête *R1*.

Pour la mise en œuvre des mécanismes de réplications, nous n'avons pas eu besoin de transmettre les requêtes selon l'ordre de leur interception. En revanche, assurer ce type d'ordre serait incontournable si nous voudrions implémenter un service de communication de groupe à base d'intercepteurs.

Bien que la norme CORBA ne prévoie pas de munir les intercepteurs d'une politique d'ordonnancement, nous pouvons concevoir une première solution basée sur la définition actuelle. En effet, nous pouvons définir un intercepteur ordonnanceur qui implémente une file d'attente et qui doit être associé en série avec d'autres intercepteurs.

Une seconde solution peut être de considérer les requêtes comme des objets de première classe, ce qui veut dire, que nous pouvons les manipuler comme des objets réguliers en les passivant, les sérialisant et les redirigeant [Taiani 2004]. Ce changement peut simplifier le développement de stratégies complexes de la tolérance aux fautes. Cependant, il demande beaucoup plus d'analyse au niveau de l'intergiciel lorsque les informations seront transmises à l'intercepteur.

Une troisième solution peut être basée sur le nouveau type d'intercepteurs que nous avons proposé dans la section 5 du chapitre III. En effet, puisque ces intercepteurs possèdent leur

propre modèle d'exécution, nous pouvons les munir d'une file d'attente et d'une stratégie de traitement des messages qui les réifient.

IV.4.3. Source de blocage et de boucles infinies

Les intercepteurs serveurs ont la particularité d'être activés à chaque fois qu'une requête arrive au niveau du squelette du serveur cible. La norme CORBA n'a pas prévu de distinguer les requêtes provenant des intercepteurs de celles qui sont émises par des clients ou encore leurs réponses respectives. Ainsi cette limite peut provoquer des situations de blocage ou des boucles infinies (voir section 1 de ce chapitre).

Pour résoudre ce problème avec les intercepteurs actuels, nous avons utilisé le contexte de la requête. En effet, lorsque l'intercepteur génère une requête, il lui associe une variable booléenne qu'il met a *true*; ensuite, lorsqu'il intercepte cette requête il peut la distinguer à partir de cette donnée.

Une seconde solution pourrait être suggérée dans le but d'améliorer la spécification actuelle des intercepteurs. En effet, l'ajout d'un nouveau champ dans le paramètre `ServerRequestInfo`, nous renseignant si la requête est générée par l'intercepteur, simplifiera le développement des mécanismes non-fonctionnels.

IV.4.4. Obligation d'invoquer chaque requête interceptée

La norme CORBA stipule qu'un intercepteur doit transmettre toute requête interceptée à un serveur cible, qui peut être le serveur original ou un serveur équivalent. Le serveur original est celui auquel le client adresse une invocation. En revanche, le serveur équivalent est un serveur qui possède la même interface IDL que le serveur original et auquel l'intercepteur, client ou serveur, redirige la requête. Pour rediriger une requête, l'intercepteur utilise l'exception `ForwardRequest` avec l'identificateur du serveur équivalent comme attribut.

L'exception `ForwardRequest` est définie comme classe finale, c'est-à-dire on ne peut pas en dériver une nouvelle classe d'exception. Ce choix de conception des intercepteurs actuels n'est pas flexible, puisque le développeur ne peut pas créer une nouvelle exception pour arrêter l'invocation.

Les seules solutions que nous avons pu mettre en oeuvre sont : la re-direction de la requête interceptée vers un serveur non-existant ou la levée de l'exception `exit(0)`. La première solution retourne l'exception `Object_Does_Not_Exist` au client et la requête ne sera pas traitée. Similairement, la seconde solution tue le brin d'exécution de la requête, ainsi la requête ne sera pas transmise et le client reçoit l'exception `Comm_Failure`.

Ces deux solutions qui ont pu résoudre le problème de l'invocation obligatoire d'une requête pour mettre en œuvre les mécanismes de réplication peuvent être une source d'erreurs. En effet, dans le cas de l'utilisation de plusieurs intercepteurs en série, la génération d'une exception au niveau d'un intercepteur active automatiquement son interception par les intercepteurs qui le précèdent. Par exemple, si notre intercepteur est placé en série avec un intercepteur de traçage, provenant d'une autre équipe de développement, la non délivrance d'une requête sera interprétée comme une erreur de communication ou d'arrêt d'un serveur.

Une troisième solution basée sur la proposition d'un nouveau type d'intercepteurs, défini dans la section 5 chapitre III, pourrait être envisagée. Comme cet intercepteur est un objet CORBA à part entière, il aura la possibilité de générer une nouvelle exception, par exemple `Request_Not_Delivered`, contenant les informations nécessaires pour la différencier des autres exceptions, à savoir `Comm_Failure` ou `Object_Does_Not_Exist`.

IV.4.5. Absence de brins propres d'exécution

Que l'objet CORBA soit mono-brin ou multi-brins, l'intercepteur serveur qui lui est associé n'est activé que lors de l'arrivée d'une requête.

Lors de l'implémentation de la réplication passive, il est logiquement nécessaire de bloquer le retour de la réponse à un client jusqu'à ce que l'intercepteur primaire envoie la requête de mise à jour de l'état au serveur répliqué. En effet, l'intercepteur perd son brin d'exécution une fois que la requête ou que la réponse est transmise à destination. Par conséquent, si l'intercepteur retourne la réponse, il n'aura aucun moyen d'envoyer la mise à jour de l'état du serveur réplique.

Par ailleurs, on ne peut pas dédier un brin à l'activité de surveillance mutuelle. Par conséquent, nous ne pouvons pas mettre en œuvre une stratégie de détection des défaillances basée sur l'envoi périodique d'un message de survie *Je suis en vie* de l'intercepteur primaire *PIS_PBRp* aux différents intercepteurs répliques *PIS_PBRb*.

En utilisant la nouvelle version des intercepteurs (présentée dans la section 5 du chapitre III), chaque intercepteur est considéré comme un objet CORBA qui possède donc son propre modèle de concurrence, et ce, de façon disjointe de celui du serveur auquel il est associé. Ainsi, dans le cas où l'intercepteur est en multi-brin et que le serveur qui lui est associé est en mono-brin, l'intercepteur peut utiliser un brin périodique pour les messages *Je suis en vie*, différent des brins propres aux autres mécanismes (tels que la transmission des requêtes ou la synchronisation de l'état des répliques). En revanche, si l'intercepteur est en mono-bin et que le serveur qui lui est associé est en multi-brin, l'intercepteur se comporte de la même manière que ceux qui répondent à la norme actuelle puisqu'il est activé lors de l'arrivée d'une requête entrante ou sortante.

IV.4.6. Incapacité de générer une réponse

Malgré le fait que les intercepteurs peuvent générer des requêtes, ils ne peuvent pas générer de réponses. Cette limite est due au fait que l'intercepteur est un pseudo objet.

Nous n'avons pu trouver de solution à ce problème avec les intercepteurs actuels. La seule façon, que nous avons mise en oeuvre, pour contourner ce problème est de traiter la requête de nouveau (voir section 1 et section 2 de ce chapitre). Ceci nous a contraint à modifier la mise en œuvre de la réplication active. En effet, lorsque le serveur principal défaille en valeur, l'intercepteur qui lui est associé redirige la requête au serveur subsidiaire pour qu'il la traite à nouveau et retourne une réponse correcte.

Avec la proposition d'intercepteurs, que nous avons définie dans la section 5 du chapitre III, les PIS seront des objets CORBA à part entière et pourraient ainsi prévoir de générer une réponse sans la traiter de nouveau.

IV.4.7. Incapacité d'utiliser la DII

Les intercepteurs actuels ne peuvent pas créer dynamiquement une requête à partir des données qu'ils récupèrent lors de l'interception d'une invocation ni même lors de son initialisation. Lors de la mise en œuvre de la réplication active, nous avons contourné le problème en passant la référence de l'interface de l'ORB dans le contexte de la première invocation (voir section 2 de ce chapitre).

Nous concevons que le passage de la référence de l'interface de l'ORB dans le paramètre `request_info` pourrait influencer les performances d'interception. La meilleure solution serait alors de le passer lors de l'initialisation de l'ORB.

Nous avons remarqué que l'initialiseur (i.e. `Initializer`) renferme l'attribut `orb_id`, de type `String`, que nous pouvons passer à l'intercepteur lors de l'appel de son constructeur. Cet attribut ne peut pas être exploité pour récupérer la référence de l'ORB. Pour pouvoir récupérer la référence de l'ORB, on doit passer cet attribut à la méthode `resolve_initial_references`. Cette méthode est définie dans l'interface de l'orb, cependant la norme pourrait l'introduire dans la définition du paramètre `Request_Info` des méthodes de l'intercepteur comme elle l'a fait au niveau du paramètre `ORBInitInfo` des méthodes de l'Initializer.

Avec la proposition d'intercepteurs, que nous avons définie dans la section 5 du chapitre III, les PIS seraient des objets CORBA à part entière et pourraient ainsi récupérer la référence de l'ORB.

IV.4.8. Incapacité de rediriger une requête à plusieurs serveurs

Les intercepteurs actuels ne peuvent utiliser que l'exception `ForwardRequest(Object)` pour rediriger la requête interceptée au serveur `Object`. La requête redirigée sera traitée par le serveur `Object`; le premier serveur cible ne traitera pas cette requête.

Lors du développement de la plate-forme DAISY, nous n'avons pas pu résoudre ce problème. Cet inconvénient, nous a empêché de mettre en œuvre la réplication active avec vote à la destination.

La seule solution possible à ce problème serait de revoir la spécification de l'exception `ForwardRequest`. En effet, au lieu d'avoir `Object` comme paramètre, il faudrait que la norme définisse un autre constructeur avec une séquence d'objets comme paramètre «`ForwarRequest(<Object>)`».

La norme FT_CORBA a défini un identificateur de groupe `IOGR`, qui est une séquence de références d'objets. Ainsi, un client utilisant un `IOGR` peut envoyer une requête à plusieurs serveurs simultanément. Comme cette solution est opérationnelle, nous estimons que sa mise en œuvre pour les intercepteurs, indispensable pour faciliter le développement des mécanismes de tolérance aux fautes, ne poserait pas de problèmes.

IV.4.9. Incapacité de modifier les paramètres de retour

Les intercepteurs standards actuels ne peuvent pas modifier les paramètres de retour d'une requête. Ces paramètres sont récupérés à partir du paramètre `RequestInfo` de la méthode activée au niveau de l'intercepteur. Les paramètres de retour sont identifiés par les champs `result` et `arguments` qui sont déclarés en lecture seule.

Nous n'avons pas pu résoudre ce problème, mais nous l'avons contourné lors du développement des mécanismes de réplication. En effet, nous avons modifié légèrement le mécanisme de réplication active afin de prendre en compte cette limitation (voir la section 3 de ce chapitre). Cependant, nous considérons que ce problème est incontournable pour d'autres mécanismes de tolérance aux fautes, à savoir le chiffrement des messages.

Cet inconvénient pourrait être résolu si la norme CORBA apportait une légère modification au niveau de la spécification du paramètre `RequestInfo`. En modifiant le mode d'accès aux champs `result` et `arguments` de `readonly` à un mode en lecture et écriture *read and write*, tout intercepteur pourrait changer le contenu des paramètres de retour d'une requête.

IV.5. MESURE DE PERFORMANCE DE LA REPLICATION PASSIVE

L'objectif des mesures de performance proposées dans cette section est de donner un ordre de grandeur du coût des mécanismes développés dans DAISY.

Pour effectuer ces mesures, nous avons développé une application de gestion de comptes bancaires et nous l'avons équipée du mécanisme de réplication passive offert par la plate-forme. Cette application consiste à fournir un service de création ou de fermeture de compte pour des clients différents. Ces clients peuvent manipuler leurs comptes en y ajoutant, ou retirant de l'argent ou en consultant leur solde. Le choix d'une telle application est guidé par le fait qu'elle possède un état, ce qui est le cas des applications auxquelles ces mécanismes de tolérance aux fautes sont destinés.

Outre le support logiciel requis pour effectuer les mesures, nous utilisons un support matériel composé d'un ensemble de machines munies de processeurs I686 de 1Ghz sous un noyau Linux 2.4 connectées via un réseau Ethernet à 100Mb/s.

Le logiciel de base utilisé correspond à la version 1.4.2 de la machine virtuelle JAVA de *SUN Microsystems* et la version 4.1.2 de l'implémentation Jorbacus de la norme CORBA développé par OOC.

Nous avons évalué le coût de l'initialisation d'une application répliquée ainsi que la durée d'une invocation répliquée. Nous avons enfin mesuré le coût relatif des différents mécanismes utilisés par la réplication passive, à savoir la sérialisation et dé-sérialisation de l'état des serveurs, l'interception des requêtes et le calcul des mécanismes non-fonctionnel.

IV.5.1. Initialisation des objets

Le Tableau 10 montre l'impact de l'utilisation des mécanismes de réplication sur l'initialisation d'un serveur d'application. Il donne la mesure de la durée de cette phase pour des applications simples (i.e. sans intercepteurs), des applications munies des mécanismes d'interception, des applications jouant le rôle de serveur réplique et des applications jouant le rôle d'un serveur primaire. Ces mesures permettent de nous renseigner sur la durée du lancement de la plate-forme DAISY et du temps nécessaire pour l'initialisation d'une nouvelle réplique dans le cas où le gestionnaire de réplication se charge du recouvrement d'erreur. Puisque nous traitons des applications distribuées, nous avons mesuré la durée d'initialisation d'une application pour deux configurations différentes. Dans la première configuration, le serveur d'application et le serveur du service de désignation coexistent sur la même machine. En revanche, dans la seconde configuration, ils se trouvent sur des machines différentes.

	Objet sans intercepteur	Objet avec intercepteur neutre	Objet secondaire	Objet primaire
Objet et serveur de désignation sur la même machine	30	34	38	53
Objet et serveur de désignation sur des machines différentes	6152	6153	6167	6189

Tableau 10. Coûts de la création des objets en *millisecondes*

Nous observons que le temps d'initialisation varie d'une manière significative suivant le choix de déploiement du serveur d'application et du serveur de désignation sur les machines hôtes. En effet, lorsque ces serveurs résident sur la même machine, le temps d'initialisation est de l'ordre de quelques dizaines de millimicrons, alors que, dans le cas où le serveur d'application et celui de désignation sont sur des machines différentes, le temps d'initialisation est de l'ordre de 6 secondes. Ceci s'explique par le coût élevé des invocations distantes à un service de désignation.

En ce qui concerne le temps d'initialisation des mécanismes de réplication, nous observons qu'il varie peu par rapport à un intercepteur neutre, en particulier lorsque les deux serveurs se trouvent sur des machines différentes.

IV.5.2. Coût des opérations

Nous avons soumis l'application bancaire à une série de 1000 expériences. Chaque expérience est composée de 1000 opérations, de retrait ou de dépôt d'argent, effectuées sur le serveur primaire. Notre mesure a concerné la durée moyenne d'une expérience.

Nous distinguons deux types d'opérations, élémentaires et non-élémentaires.

En effet, une opération est composée d'une opération de recherche du titulaire du compte puis d'une opération d'addition ou de soustraction qui est équivalente à une mise à jour du contenu du compte. Dans le cas d'une opération élémentaire, nous avons initialisé la banque avec un seul compte, l'opération de recherche est immédiate et le temps d'exécution de la requête est très court.

Dans le cas d'une opération non-élémentaire, nous avons augmenté le temps de traitement fonctionnel d'une requête, en particulier le temps de recherche d'un compte, car nous initialisons le serveur bancaire avec plusieurs comptes. Nous utilisons ce second type d'invocations pour voir si le coût de la tolérance aux fautes est fonction du traitement fonctionnel ou pas.

Sur la Figure 53, les deux premiers cylindres représentent les durées respectives d'une opération élémentaire dans deux cas de figures :

1)- traitement sans utilisation de mécanisme de tolérance aux fautes;

2)- traitement lorsque l'application est munie du mécanisme de réplication passive.

Les deux seconds cylindres montrent les mesures des opérations non-élémentaires ces deux mêmes cas de figure.

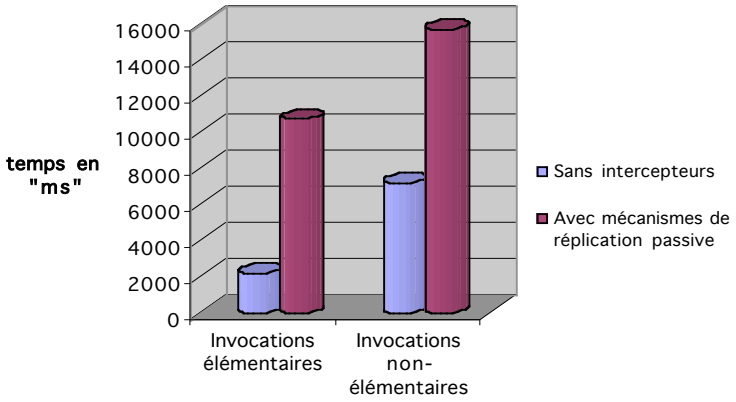


Figure 53. Durée moyenne d'une expérience de 1000 invocations d'un service bancaire

Les deux premiers cylindres de la figure ci-dessus montrent que la durée d'une opération répliquée est de 8 ms (800 ms pour 1000 invocations) de plus qu'une opération simple. Dans ce cas, la durée d'une invocation élémentaire répliquée est 4 fois plus élevée que celle d'une invocation élémentaire simple.

Par ailleurs, les deux seconds cylindres de la Figure 53 montrent que la différence entre une opération simple et une opération répliquée n'a pas augmenté, elle est restée égale à 8ms. En revanche, la durée d'une invocation répliquée ne représente plus que le double de la durée d'une invocation simple. Par conséquent, nous pouvons conclure que le coût de la tolérance aux fautes est constant. En substance, le coût relatif de la tolérance aux fautes au sein d'une application dépend de la complexité et donc de la durée du traitement non-fonctionnel.

IV.5.3. Coût des différents mécanismes

Sur la Figure 54, nous analysons la répartition du coût des mécanismes élémentaires introduits par la plate-forme DAISY. Les mécanismes élémentaires nécessaires à la mise en œuvre des mécanismes de tolérance aux fautes par réplication sont les suivants: interception, protocole de réplication, sérialisation et dé-sérialisation.

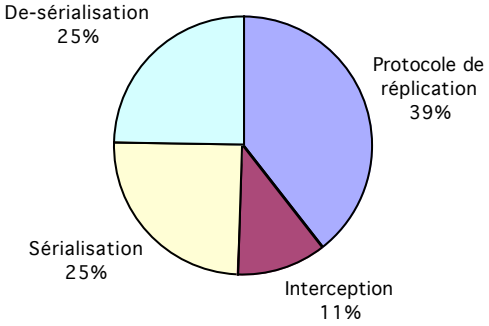


Figure 54. Répartition des différents mécanismes offerts par DAISY

Nous remarquons qu'il existe deux familles de mécanismes : les mécanismes à coût fixe et les mécanismes à coût variable. La première famille correspond au mécanisme d'interception, et au protocole, en particulier à l'analyse des contextes des requêtes et à la génération des requêtes de synchronisation de l'état. La seconde famille renferme les mécanismes de sérialisation et de dé-sérialisation de l'état d'une application.

Dans la première famille, les mécanismes d'analyse du contexte des requêtes et de la génération des requêtes de synchronisation représentent 39 % de la durée des mécanismes de réplication, alors que les mécanismes d'interception en représentent que 11 %. Par conséquent, le coût constant représente 50% du temps d'exécution des mécanismes de réplication.

Dans la seconde famille, les mécanismes de récupération de l'état et de la mise à jour représentent chacun 25 % du temps de l'ensemble des mécanismes de réplication. Ces mécanismes sont fortement liés à la taille de l'état de l'application. Bien que cette durée représente 50 % de la durée des mécanismes de réplication, dans nos expériences, elle peut varier d'une application à une autre.

IV.6. CONCLUSION

Nous avons présenté dans ce chapitre la mise en œuvre de la variante à composant intégrés de la plate-forme DAISY. Cette mise en œuvre montre que :

- le déploiement de la tolérance aux fautes en utilisant les intercepteur CORBA est transparent par rapport aux utilisateurs et aux développeurs de l'application.

- les intercepteurs CORBA ne sont pas appropriés au développement de mécanismes de tolérance aux fautes complexes.

- la spécification des intercepteurs CORBA doit évoluer pour mieux s'adapter aux besoins de la tolérance aux fautes et de la prise en compte des traitements de groupe.

Dans un premier temps, nous avons présenté l'implémentation de la réplication passive. Nous avons détaillé les différents mécanismes présents dans le composant de communication client/serveur *CC-C/S* (i.e. intercepteur client, *PIC*) et le composant de réplication passive *PBR* (i.e. intercepteur serveur, *PIS_PBR*). Par exemple, le mécanisme de détection de défaillance, le mécanisme du recouvrement d'erreurs, le mécanisme de capture des points de reprises, etc. Nous avons montré que l'implémentation de tels mécanismes n'était pas facile à mettre en œuvre à cause des restrictions et de la non-adaptation des intercepteurs. Nous avons dégagé de cette première mise en œuvre 6 limites : le manque d'information à l'initialisation, l'absence de politique d'ordonnancement des messages, la non-distinction des requêtes qu'il génère et de leurs réponses, l'obligation de transmettre la requête interceptée et l'absence de brin d'exécution.

Dans un second temps, nous avons présenté l'implémentation de la réplication active. Nous avons détaillé les différents mécanismes présents dans le composant de réplication active *ACR* (i.e. intercepteur serveur, *PIS_ACR*), par exemple, le mécanisme de diffusion des messages, de vote et du recouvrement d'erreur par arrêt du serveur et par valeur. Suite à cette mise en œuvre, nous avons dégagé 4 limites pour les intercepteurs actuels : l'intercepteur ne possède pas de moyen standard pour créer des requêtes dynamiques, il ne peut pas assurer la réplication active avec vote à la destination, il ne peut pas changer le contenu d'une réponse et il ne peut pas la créer.

Ensuite, nous avons présenté l'implémentation des composants utilitaires, à savoir l'usine à objets et le gestionnaire de réplication. Nous avons montré qu'il existe deux manières pour implémenter l'usine à objet. La première se base sur l'aspect langage et la seconde se base sur l'aspect service. Nous avons montré l'intérêt de la seconde approche pour fournir une plate-forme indépendante des applications qui l'utilisent. Aussi, nous avons présenté l'implémentation du gestionnaire de réplication qui permet le déploiement de la plate-forme DAISY et se charge de la dernière partie de la gestion de la défaillance d'un serveur (i.e. la mise en fonction d'un nouveau serveur).

Par la suite, nous avons repris les problèmes que nous avons relevés lors de la phase d'implémentation de la plate-forme DAISY. Nous avons identifié l'origine de chaque problème et nous avons discuté, dans la mesure du possible, trois types de solutions. Le premier type de solution se base sur la spécification actuelle. Le second a pour objectif d'apporter une légère modification de la spécification actuelle. Tandis que, le troisième se base sur un nouveau type d'intercepteur que nous jugeons approprié pour la mise en œuvre des mécanismes de tolérance aux fautes.

Enfin, nous avons synthétisé les résultats que nous avons obtenus lors de l'expérimentation de la plate-forme DAISY, dans le cas de la réplication passive et celui de la réplication active. Nous avons noté que le coût des mécanismes de réplication passive est relativement constant aussi bien lors de l'initialisation de la plate-forme que lors du déroulement de ces traitements.

CONCLUSION GENERALE

Cette thèse a traité de l'étude du potentiel réflexif des intergiciels standards, tout particulièrement le bus à objets de la norme CORBA dans le but de l'exploiter pour munir les applications distribuées de mécanismes de tolérance aux fautes.

La réflexivité est un moyen puissant pour réaliser un découplage entre les aspects applicatifs et les aspects non-fonctionnels d'un système. Profitant de ce pouvoir de découplage (appelé aussi *cross-cutting concerns*), la réflexivité permet de munir les applications, en particulier celles qui sont distribuées, de mécanismes de tolérance aux fautes séparés de leur code fonctionnel.

Le développement des applications distribuées est facilité par l'utilisation des intergiciels, qui servent d'intermédiaires entre les applications et le système d'exploitation. Si ces intergiciels ont réussi à simplifier la programmation distribuée et ont amélioré la portabilité des applications, ils n'offrent pas toujours de solutions simples pour la prise en compte de la tolérance aux fautes.

Jusqu'à tout récemment, les solutions proposées pour pallier ce problème pouvaient être classées en deux catégories : classique et réflexive. Dans la première catégorie, on s'attaque directement au problème de la mise en œuvre des mécanismes de tolérance aux fautes sans se soucier de l'interopérabilité de la solution (i.e. approche par intégration), de sa transparence par rapport au développeur de l'application (i.e. approche par service), ou encore de sa portabilité (i.e. approche par interception). Dans la seconde catégorie, l'intérêt est lié à la séparation des mécanismes de tolérance aux fautes des mécanismes fonctionnels. Les mécanismes de tolérance peuvent se trouver intégrés à l'ORB (i.e. OpenORB, Dynamic TAO et OpenCORBA) ou localisés dans des composants externes à l'intergiciel (i.e. FRIENDS). Ces deux catégories n'offrent pas de solutions standards pour la prise en compte de la tolérance aux fautes. S'allignant avec les approches classiques, les travaux effectués dans le cadre de la norme FT-CORBA ont abouti à une solution standard. Cependant, son utilisation reste relativement complexe et sa mise en œuvre lourde et non transparente pour les développeurs des applications.

Nous avons proposé une architecture simple utilisant des mécanismes réflexifs standards, à savoir les intercepteurs CORBA et la sérialisation JAVA, pour fournir des mécanismes de tolérance aux fautes à des applications distribuées. Cette architecture comporte deux ensembles de composants : les composants de tolérance aux fautes et les composants utilitaires. Les premiers fournissent des mécanismes de réplication classiques et les seconds assurent la gestion de leur mise en œuvre.

Nous avons pu montrer que cette architecture peut être mise en œuvre de deux manières différentes puisqu'elle permet de fournir des composants de tolérance aux fautes externes ou intégrés à l'intergiciel CORBA.

- Pour que les composants de tolérance aux fautes soient placés à l'extérieur de l'ORB, nous avons considéré les intercepteurs CORBA comme des éléments de base pour construire un protocole à méta-objet. Les mécanismes de réplication sont alors placés au niveau des objets CORBA. Si cette approche est similaire à l'approche classique par service, elle n'implique pas le développeur de l'application dans la mise en œuvre de la tolérance aux fautes puisque ce rôle est assuré par le protocole à méta-objet qui est décrit au niveau des intercepteurs CORBA.

- Pour que les composants de tolérance aux fautes soient intégrés à l'ORB, nous avons considéré les intercepteurs CORBA comme des méta-objets à part entière. Par conséquent, les mécanismes de réplications sont définis au niveau des intercepteurs. Si cette approche est similaire à l'approche classique par intégration, elle n'altère pas l'interopérabilité de l'intergiciel car elle ne modifie pas son code.

Nous avons montré aussi que la spécification actuelle des intercepteurs CORBA est limitée pour munir les applications réparties des mécanismes classiques de tolérance aux fautes. Nous avons classé ces limites en deux grandes familles : les limites conceptuelles et les limites pratiques.

Les limites conceptuelles s'articulent particulièrement autour de la réflexivité des intercepteurs : est ce qu'ils permettent de fournir un protocole à méta-objets ? peuvent-ils jouer le rôle de méta-objets ?

Nous avons montré qu'en tant que composants élémentaires pour la construction d'un MOP, les intercepteurs peuvent notifier un méta-objet de la création ou la destruction de l'objet de base. En revanche, ils sont incapables de fournir la référence de ce dernier et d'assurer un lien pour observer aussi bien l'objet de base que son interface. En outre, les intercepteurs actuels ne peuvent pas assurer un lien pour que le méta-niveau modifie la structure et l'interface de l'objet de base. Lorsqu'un méta-objet et un objet sont liés par le biais des intercepteurs, la re-direction des requêtes ne peut être réalisée que vers des méta-objets ayant la même interface que celle de l'objet cible. Cet inconvénient influence la mise en œuvre des composants de tolérance aux fautes, puisque leur interface ne peut pas être standard pour toutes les applications mais doit être fonction de l'application.

Puisque les intercepteurs actuels sont des pseudo-objets CORBA, c'est-à-dire ne possèdent pas de références et leur interface n'est pas invocable, il n'est pas envisageable de leur associer des intercepteurs (i.e. pas de notion de méta-intercepteurs). La norme CORBA ne permet qu'une association en série des intercepteurs. Par exemple, un client voulant s'adresser à deux applications différentes, munies de deux mécanismes non-fonctionnels différents, voit sa requête à destination du premier serveur subir, quand bien même, le traitement non-fonctionnel du second. Ceci engendre des incohérences lorsqu'on utilise deux mécanismes

non-fonctionnels incompatibles. Aussi, nous avons remarqué que le cycle de vie de l'intercepteur est lié à celui de l'application à laquelle il est associé. Ainsi, la défaillance de l'application engendre l'arrêt de son intercepteur. Ceci élargit la zone de confinement des erreurs et limite le pouvoir des intercepteurs en tant que méta-objets.

Les limites pratiques concernent des problèmes que nous avons rencontrés lors de la mise en œuvre des mécanismes de réplication au niveau des intercepteurs. Ces limites s'articulent autour de l'observation et du contrôle des requêtes, de l'observation des objets et des moyens propres des intercepteurs. En effet, les limites relatives au contrôle des requêtes sont liées à l'absence d'une politique d'ordonnancement, à l'incapacité des intercepteurs de générer une réponse ou de modifier son contenu et à leur incapacité de diffuser la requête à plusieurs serveurs. Les intercepteurs ont un pouvoir d'observation des requêtes limité car ils ne peuvent pas distinguer celles qui sont générées par un client de celles qui qu'il a généré lui-même. De plus, les intercepteurs sont limités au niveau de l'observation des serveurs auxquels ils sont associés, puisqu'ils ne peuvent pas les identifier. En termes de moyens propres, les intercepteurs ne possèdent pas leur propre brin d'exécution et ne peuvent pas récupérer la référence de l'interface de l'ORB dans lequel ils sont initialisés.

Malgré les limites conceptuelles et pratiques des intercepteurs, nous avons pu implémenter un prototype de la variante à composants de tolérance aux fautes intégrés de la plate-forme DAISY. Ce prototype qui fournit aussi bien le mécanisme de réplication passive que celui de la réplication active, prouve que les mécanismes standards, c'est-à-dire les intercepteurs CORBA et la sérialisation JAVA, peuvent fournir des COTS standards, transparents et potables assurant la tolérance aux fautes.

Des mesures effectuées sur cette plate-forme ont montré que la durée d'exécution des mécanismes de réplication est indépendante de l'application à laquelle elle est associée. La seule variation concerne le temps de récupération et de la mise à jour de l'état, fournis par le mécanisme de sérialisation JAVA. Cette durée est fonction de la taille de l'état de l'application.

Grace à une analyse basée sur notre expérience, nous avons proposé une nouvelle génération d'intercepteurs mieux adaptés à la mise en œuvre des mécanismes non-fonctionnels et en particulier de réplication. L'idée principale est de transformer ces pseudo-objets non adressables et sans brin d'exécution propre en des objets intégrés adressables et possédant leur propre brin d'exécution.

En conclusion, les travaux menés au cours de cette thèse permettent de prouver l'intérêt que présentent les mécanismes réflexifs standards au niveau des intergiciels pour fournir de la tolérance aux fautes aux applications distribuées. En particulier, ces mécanismes permettent de fournir la réplication des services d'une manière standard, portable et transparente. Néanmoins, la spécification actuelle des mécanismes réflexifs CORBA ne permet pas une mise en œuvre simple et efficace de la tolérance aux fautes, ce qui nécessite donc des améliorations dans le sens de celles que nous avons proposées.

Comme perspectives de notre travail, il serait intéressant et sans doute relativement aisé de traiter la problématique de changement de mécanisme de réplication en fonction des ressources à disposition. Ceci pourrait certainement affiner l'analyse que nous avons effectuée et peut-être identifier de nouvelles limites, dans le but d'améliorer notre proposition pour la nouvelle génération d'intercepteurs.

Cette thèse montre aussi comment améliorer les intercepteurs CORBA pour construire un protocole à méta-objets. La comparaison entre la proposition qui vise à promouvoir les intercepteurs en méta-objets et celle qui sera à l'origine de la naissance d'un protocole à méta-objet intégré au niveau de l'intergiciel pourrait influencer les nouvelles directions de la norme CORBA.

BIBLIOGRAPHIE

[Agha 1993] G. Agha, S. Frolund, R. Panwar et D. Sturman, “A Linguistic Framework for Dynamic Composition of Dependability Protocols”, in *the IFIP conference on Dependable Computing for Critical Applications (DCCA-3),* (Palermo, Italie), pp.197-207, 1993.

[Arlat 2000] Jean Arlat, Jean-Paul Blanquart, Thierry Boyer, Yves Crouzet, Marie-Hélène Durand, Jean-Charles Fabre, Michel Founau, Mohamed Kaâniche, Karama Kanoun, Philippe Le Meur, Corrine Mazet, David Powell, François Scheerens, Pascale Thévenod-Fosse et Hélène Waeselynch, *Composants logiciels et sûreté de fonctionnement,* 158p., Hermes, Paris, 2000.

[Ashish 1997] Singhai Ashish, Aamod Sane et Roy Campbell, “Reflective ORBs : Supporting Robust, Time-Critical Distribution”, in *Workshop on Reflective Real-Time Object Oriented Programming and Systems (ECOOP'97),* (Jyvaskyla, Finlande), 1997.

[Ashish 1998] Singhai Ashish, Aamod Sane et Roy H. Campbell, “Quarterware for Middleware”, in *IEEE International Conference on Distributed Computing Systems (ICDCS),* (Amsterdam, Pays Bas), 1998.

[Avizienis 2004] Algidras Avizienis, Jean-Claude Laprie, Brian Randell et Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Scure Computing”, *IEEE Transactions on Dependable and secire computing*, 1 (1), pp.11-33, Janvier-Mars 2004.

[Baldoni 2002] R. Baldoni, C. Marchetti et A. Termini, “Active Software Replication Through Three-tier Approach”, in *Symposium on Reliable Distributed Systems (SRDS),* (Osaka, Japon), 2002.

[Baldoni 2003] R. Baldoni, C. Marchetti et L. Verdi, “CORBA Request Portable Interceptors: Analysis and Application”, *Concurrency and Computation Practice & Experience*, 15 (6), pp.551-579, 2003.

[Bennani 2003] Mohamed Taha Bennani, “Systemes sûrs de fonctionnement a base d'intergiciels reflexifs. Mecanismes et architecture du banc de test”, in *4eme Congres de l'Ecole Doctorale Systemes (EDSYS),* (Toulouse, France), octobre 2003.

[Bennani 2004] Mohamed Taha Bennani, Laurent Blain, Ludovic Courtes, Jean-Charles Fabre, Marc-Olivier Killijian, Eric Marsden et François Taïani, “Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization”, in *International Conference on Dependable Systems and Networks (DSN 2004),* (Florence, Italie), pp.549-554, juin 2004.

[Bershad 1995] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers et Craig Chambers, “Extensibility, Safety and Performance in the SPIN Operating System”, in *15th ACM Symposium on Operating Systems*

Principles, (Copper Mountain Resort, Colorado, Etats-unis), Décembre 1995.

[Birman 1993] K. P. Birman, "The Process Group Approach to Reliable Distributed Computing", *Communications of the ACM*, 36 (12), pp.36-53, décembre 1993.

[Birman 1996] Kenneth P. Birman, *Building Secure and Reliable Network Applications,* 591p., 1996.

[Birrell 1984] Andrew D. Birrell et Bruce Jay Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, 2, pp.39-59, février 1984.

[Chandra 1996] Tushar Deepak Chandra, Vassos Hadzilacos et Sam Toueg, "The Weakest Failure Detector for Solving Consensus", *Journal of the ACM*, 43 (4), pp.685-722, juillet 1996.

[Cheriton 1985] D. R. Cheriton et W. Zwaenepoel, "Distributed Process Groups in the V Kernel", *ACM Transactions on Computer Systems*, 3 (2), pp.77-107, mai 1985.

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

[Chiba 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),* (Kaiserslautern, Allemagne), pp.482-501, juillet 1993.

[Clarke 2001] Michael Clarke, Gordon S. Blair, Geoff Coulson et Nikos Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware", in *International Conference on Distributed Systems Platforms,* (Heidelberg - Allemagne), pp.160-178, Springer-Verlag Londres, grande-bretagne, 2001.

[Cointe 1987] Pierre Cointe, "Metaclasses are First Class: the ObjVlisp Model", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications, OOPSLA,* (Orlando, Etats-unis), pp.156-167, ACM Sigplan Notices, octobre 1987.

[Creasy 1981] C. J. Creasy, *"The Origin of the VM/370 Time-Sharing System",* IBM J. Research and Development, 25 (5), 1981.

[Cristian 1996] Flaviu Cristian, "Synchronous and Asynchronous Group Communication (Long Version)", Personal Communication, avril 1996.

[Engler 1995] D. R. Engler, M. F. Kaashoek et J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", in *15th ACM Symposium on Operating System Principles (SOSP'95),* (Cooper Mountain Resort, Colorado, Etats-unis), pp.251-266, 1995.

[Ezhilchelvan 1995] Paul. D. Ezhilchelvan, Raimundo. A. Macedo et Santosh. k. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol", in *5th International Conference on Distributed Computing Systems (ICDCS'95),* (Vancouver (BC), Canada), IEEE Comuter Society Washington, DC, Etats-unis, mai 1995.

[Fabre 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*, 47 (1), pp.78-95, janvier 1998.

[Felber 1998a] P. Felber, *The CORBA Object Group Service,* Ecole Polytechnique Fédérale de Lausanne, 1998a.

[Felber 1996] P. Felber, B. Garbinato et R. Guerraoui, "The Design of a CORBA Group Communication Service", in *15th IEEE Symposium on Reliable Distributed Systems,* pp.150-159, octobre 1996.

[Felber 1998b] P. Felber, R. Guerraoui et A. Schiper, "The implementation of a CORBA object group service", *Theory and Practice of Object Systems*, 4 (2), pp.93-105, avril 1998b.

[Fetzer 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, Etats-unis), pp.282-291, 1997.

[Fischer 1985] M. J. Fischer, N. A. Lynch et M. S. Paterson, "Impossibility of Distributed Consensus With One Faulty Process", *Journal of the ACM*, 32 (2), pp.374-382, avril 1985.

[Floyd 1997] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne et Lixia Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing", *IEEE/ACM Transactions on Networking (TON)*, 5 (6), pp.784-803, décembre 1997.

[Foote 1989] Brian Foote et Ralph E. Johnson, "Reflective Facilities in Samlltalk-80", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications, OOPSLA,* (A. S. Notices, Ed.), (New Orleans, Etats-unis), pp.327-335, octobre 1989.

[Gamma 1997] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, *Design Patterns,* Professional Computing, 395p., 1997.

[Golberg 1983] A. Golberg et D. Robson, *Smalltalk-80: The langage and its Implementation*, Reading, 1983.

[Golm 1998] Michael Golm et Jurgen Kleinoder, "MetaXa and the Future of Reflection", in *ACM Conference on Object Oriented Programming Systems, Languages, and Applications, OOPSLA,* (Vancover, Canda), octobre 1998.

[Guerraoui 1997] R. Guerraoui, B. Garbinato et K.Mazouni, "GARF: A Tool for Programming Reliable Distributed Applications", *IEEE Concurrency*, 5 (4), pp.32-39, 1997.

[Henning 1999] Michi Henning et Steve Vinoski, *Advanced CORBA Programming with C++,* Professional Computing, 1999.

[IONA 1994] IONA, *An Introduction to ORBIX+ISIS*, IONA Technologies Ltd and Isis Distributed Systems, Inc, 1994 1994.

[IONA 2000a] IONA, "Filtering Operation Calls", in *Orbix Programmer's Guide C++ Edition* (I. T. PLC, Ed.), pp.319-338,2000a.

[IONA 2000b] IONA, “Loading Objects at Runtime”, in *Orbix Programmer's Guide C++ Edition* (I. T. PLC, Ed.), pp.367-388,2000b.

[IONA 2000c] IONA, “Transforming Requests”, in *Orbix Programmer's Guide C++ Edition* (I. T. PLC, Ed.), pp.401-408,2000c.

[Karablieh 2002] F. Karablieh et R. A. Bazzi, “Heterogeneous Checkpointing for Multithreaded Applications”, in *21st Symposium on Reliable Distributed Systems (SRDS'02),* pp.140-149, Osaka, Japon, 2002.

[Kiczales 1991] Gregor Kiczales, Jim des Rivières et Daniel G.Bobrow, *The Art of the Metaobject Protocol,* MIT Press, Londres, Angleterre, 1991.

[Killijian 2000] M.-O. Killijian, *Tolérance aux fautes sur CORBA par protocole à métaobjets et langages réflexifs,* Informatique, Institut National Polytechnique, 2000.

[Killijian 2002] M.-O. Killijian, J.-C. Ruiz-Garcia et J.-C. Fabre, “Portable serialization of CORBA objects: a Reflective Approach”, in *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA,* (Seattle, Washington, Etats-unis), 2002.

[Kon 2000] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã et Roy H. Campbell, “Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB”, in *IFIP/ACM International Conference on Distributed systems platforms,* (New York, Etats-unis), pp.121 - 143, 2000.

[Lamport 1982] L. Lamport, R. Shostak et M. Pease, “The Byrantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 4 (3), pp.382-401, juillet 1982.

[Laprie 1995] 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,* 369p., 1995.

[Lindholm 1999] Tim Lindholm et Frank Yellin, *The Java Virtual Machine Specification Second Edition,* The Java Series, 496p., 1999.

[Maes 1987a] Pattie Maes, *Computational Reflection,* Vrije Universiteit, 1987a.

[Maes 1987b] Pattie Maes, “Concepts and Experiments in Computational Reflection”, in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* (Orlando, Florida, Etats-unis), pp.147-155, octobre 1987b.

[Maffeis 1995] Silviano Maffeis, *Run-Time Support for Object-Oriented Distributed Programming,* Université de Zurich, 1995.

[Microsoft 2004] Microsoft, "FxCop", http://www.gotdotnet.com/team/fxcop,

[Moser 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, 39 (4), 1996.

[Moser 1999] L. E. Moser, P. M. Melliar-Smith et P. Narasimhan, "A Fault Tolerant Framework for CORBA", in *29th International Symposium on Fault-Tolerant Computing,* (Madison, Wisconsin, Etats-unis), 1999.

[Narasimhan 2001] Priya Narasimhan, Louise E. Moser et P. M. Melliar-Smith, "State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects", in *The International Conference on Dependable Systems and Networks (DSN'2001),* (Washington, DC, Etats-unis.), pp.261-270, IEEE Computer Society, 2001.

[Nelson 1991] Greg Nelson, *Systems Programming with Modula-3,* 267p., 1991.

[Neumann 1956] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components", in *Automata Studies* (C. E. Shannon and J. McCarthy, Eds.), pp.43-98,Princeton University Press, 1956.

[OMG 2002a] Object Management Group. OMG, *Common Object Request Broker Architecture: Core Specification,* 1150p., 2002a.

[OMG 2002b] Object Management Group. OMG, "Fault Tolerant CORBA", in *Common Object Request Broker Architecture: Core Specification* OMG formal/02-12-06, pp.939-1043,2002b.

[OMG 2002c] Object Management Group. OMG, "Portable Interceptors", in *Common Object Request Broker Architecture: Core Specification* OMG formal/02-12-06, pp.789-853,2002c.
[Powell 1991] D. Powell, *Delta-4,* 1, Springer-Verlag, 1991.

[Powell 1992] D. Powell, "Failure Mode Assumptions and Assumption Coverage", in *International Symposium on Fault-Tolerant Computing (FTCS),* (Boston, MA , Etats-unis), pp.386-395, IEEE, 1992.

[Powell 1998] D. Powell, "Distributed Fault Tolerance: A Short Tutorial", in *IFIP International Workshop on Dependable Computing and its Applications,* (Johannesbourg, Afrique du Sud), pp.1-12, 1998.

[Roman 1999] M. Roman, F. Kon et R. H. Campbell, "Design and Implementation of Runtime Reflection in Communication Middleware: The DynamicTAO Case", in *19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware.,* (Austin, Texas, Etats-unis), pp.122-127, 1999.

[Ruiz-García 2003] J.-C. Ruiz-García, M.-O. Killijian, J.-C. Fabre et P. Thévenod-Fosse, "Reflective Fault-Tolerant Systems: From Experience to Challenges", *IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems*, 52, pp.237-254, 2003.

[Saikoski 2003] Katia Saikoski et Geoff Coulson, "Supporting Dependable Distributed Applications Through a Component-Oriented Middleware-Based Group Service", in *Workshop on Software Architectures for Dependable Systems (WADS),* (Portland, Oregon, Etats-unis), pp.99-122, 2003.

[Schmidt 1999] Douglas C. Schmidt et Chris Cleeland, *"Applying Patterns to Develop*

Extensible ORB Middleware", IEEE Communications Magazine, 37 (4), 1999.

[Smith 1982] Brian Cantwell Smith, *Procedural Reflection in Programming Languages,* Massachussetts Institute of Technology, 1982.

[Smith 1984] Brian Cantwell Smith, "Reflection and Semantics in LISP", in *11th Annual Symposium on Principles of Programming Languages,* (Salt Lake City, Utah, Etats-unis), pp.23-35, 1984.

[Steinke 1995] Steve Steinke, *"Middleware Meets the Network",* LAN: The Network Solutions Magazine, 10 (13), 1995.

[Sun 1998] Sun Microsystems corporation Sun, *Java Object Serialization Specification*, Technical Report, november 1998.

[Sun 2004] Sun Microsystems corporation Sun, *Java Remote Method Invocation Specification,* 121p., 2004.

[Taiani 2004] F. Taiani, *La réflexivité dans les architectures multi-niveaux,* Paul Sabatier de Toulouse, 2004.

[Verissimo 1990] P. Verissimo et J. Marques, "Reliable Broadcast for Fault-Tolerance on Local Computer Networks", in *IEEE Symposium on Reliable Distributed Systems (SRDS-9),* (Huntsville, Alabama, Etat-unis.), pp.54-63, 1990.

[Wade 1993] Andrew E. Wade, "Single Logical View Over Entreprise-Wide Distributed Databases", in *ACM SIGMOD International Conference on Management of Data,* (Washington, D.C., Etats-Unis), pp.441-444, mars 1993.

[Wang 2001] Nanbor Wang, Kirthika Parameswaran et Douglas Schmidt, "The Design an Performance of Meta-Programming Mechanisms for Object Request Brocker Middleware", in *USENIX,* (Boston, Massachussets, Etats-unis.), juin 2001.

[Welch 1999] Ian Welch et Robert J. Stroud, "Kava - A Reflective Java Based on Bytecode Rewriting", in *OOPSLA Workshop on Reflection and Software Engineering,* (Denver, Colorado, Etats-unis), pp.155-167, Springer-Verlag, 1999.

[Williams 1994] Sara Williams et Charlie Kindel, "The Component Object Model", http://msdn.microsoft.com/library/default.asp?url=/library/enus/dncomg/html/msdn_comppr.asp, 2004

[Youngblood 2004] G. Michael Youngblood, Dinane Cook et Sajal Das, "Middleware", in *Smart Environments: Technology, Protocol and Applications* Parallel and Distributed Computing, Wiley, 2004.