Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Un framework de monitoring automatique d'objets distribués en CORBA

Aussems, Patrick

Award date:
2002

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. Dec. 2022

Un framework de monitoring automatique d'objets distribués en CORBA

Patrick Aussems

Facultés Universitaires Notre-Dame de la Paix

Namur

Septembre 2002

Résumé

Les systèmes distribués prennent une place de plus en plus importante dans l'informatique contemporaine, il en va de même pour les activités de monitoring qui permettent de vérifier leur bon fonctionnement et d'étendre leurs fonctionnalités. Dans ce mémoire nous vous proposons un framework qui repose sur CORBA et ses intercepteurs portables. Il permet le développement de moniteurs distribués et peut être intégré avec un minimum de modifications à des applications existantes.

Abstract

Distributed systems take an increasingly significant place in contemporary data processing, and so do the monitoring activities which make it possible to check their correctness and to extend their functionalities. In this thesis we propose a framework relying on CORBA and its portable interceptors which allows the development of distributed monitors and can be integrated into existing applications with a minimum of changes.

Remerciements

Mes premiers remerciements vont à mon promoteur, Monsieur Vincent Englebert, pour ses conseils, ses remarques constructives ainsi que pour son encouragement tout au long de la rédaction de ce mémoire.

Merci également à tous les chercheurs du laboratoire *Intelligent Transportation Systems* du MIT qui m'ont permis de passer un très agréable stage de fin d'études, et grâce à qui j'ai eu l'occasion de me familiariser avec l'utilisation de CORBA dans un projet de grande envergure.

Merci à mes parents qui m'ont aidé à financer ce stage et qui ont passé de nombreuses heures à comprendre ce mémoire et à le relire afin d'en améliorer la lisibilité.

Merci à tous mes amis qui m'ont encouragé pendant les derniers mois qui ont précédé la remise de ce mémoire. Je remercie plus particulièrement ceux qui n'ont pas hésité à consacrer un peu de temps libre pour m'aider à en améliorer la qualité.

Merci aussi à l'ensemble de la communauté du logiciel libre qui m'a offert la possibilité de réaliser ce mémoire au moyen d'outils gratuits et d'excellente qualité.

Merci d'avance à toutes les personnes qui liront ce mémoire et qui viendront assister à sa défense.

Patrick Aussems

Table des matières

1 CORBA **17**
1.1 Les objets CORBA . . . 17
1.1.1 La description des objets . . . 18
1.1.2 Exemple de description IDL . . . 19
1.1.3 L'interface statique . . . 20
1.1.4 L'interface dynamique . . . 20
1.1.5 Les références d'objets . . . 21
1.2 Le bus CORBA . . . 21
1.3 Localisation des composants . . . 22
1.4 Les intercepteurs CORBA . . . 22
1.4.1 Intercepteurs au niveau du client . . . 24
1.4.2 Intercepteurs au niveau du serveur . . . 24
1.4.3 Ordre d'activation des intercepteurs . . . 26
1.4.4 Exemples d'applications des intercepteurs . . . 27

2 Monitoring **31**
2.1 Les fonctions du monitoring . . . 32
2.1.1 Vérification de la conformité aux spécifications . . . 32
2.1.2 Contrôle . . . 33
2.2 Difficultés liées au monitoring de systèmes distribués . . . 34
2.3 Les mécanismes d'interception . . . 35
2.3.1 Le sniffing sur le réseau . . . 35
2.3.2 Les intercepteurs au niveau de l'OS . . . 36
2.3.3 L'instrumentation des stubs et des skeletons . . . 36
2.3.4 L'utilisation de proxy . . . 37
2.3.5 Les intercepteurs CORBA . . . 37

3 Étude de l'existant **39**
3.1 CorbaTrace . . . 39
3.2 MODOCC - MOnitoring of Distributed Object and Component Communication . . . 41
3.3 Architecture proposée dans le Java Developper's Journal . . . 42

4 Architecture **45**
4.1 Objectifs 45
4.2 Hypothèses de travail 47
4.3 Description générale de l'architecture 47
4.4 Choix du procédé d'interception 48
4.5 Description des composants 50
4.5.1 L'intercepteur au niveau du serveur 50
4.5.2 L'accès à la configuration des intercepteurs 52
4.5.3 Les moniteurs 53
4.5.4 Le dispatcheur 55
4.6 Enregistrement des intercepteurs et des moniteurs 57
4.6.1 L'enregistrement d'un intercepteur 58
4.6.2 L'enregistrement d'un moniteur 59

5 Prototype **61**
5.1 Choix de l'implémentation CORBA 61
5.2 Implémentation 62
5.3 Mode d'emploi 62
5.3.1 Adaptation du serveur 63
5.3.2 Les moniteurs 64
5.4 Limitations du prototype 70
5.5 Évaluation des performances 70
5.5.1 Mesures 70
5.5.2 Résultats 71

6 Discussion **73**
6.1 Évaluation et possibilités d'amélioration 73
6.1.1 Les modifications nécessaires 73
6.1.2 L'interactivité des moniteurs 74
6.1.3 Performances 75
6.2 Solutions alternatives 76

A Code source **81**
A.1 ORBInitializer 81
A.2 ServerRequestInterceptor 83
A.3 Accessor 89
A.4 Dispatcher 91
A.5 Monitor 99

B Étude de cas : le serveur d'impression **101**
B.1 PrintServer 101
B.2 PrintServerMonitor 104

Liste des tableaux

1.2 Informations disponibles pour les points d'interception serveur 25

4.2 Caractéristiques des différentes techniques d'interception 49

5.2 Délais d'invocation 71
5.4 Délais d'invocation, moniteurs fictifs 72

Table des figures

1.1 Points d'interception 23
1.2 Ordre d'activation pour les intercepteurs du client 27

3.1 Monitoring : Visualisation d'un fichier XMI à l'aide de MagicDraw UML 40
3.2 MODOCC : Visualisation des interactions dans un système distribué 43

4.1 Architecture 48
4.2 Base de données des interfaces et moniteurs 56
4.3 Monitoring d'une requête 57
4.4 Monitoring de la réponse 58
4.5 Initialisation d'un intercepteur 59
4.6 Initialisation d'un moniteur 60

5.1 Machine à états du serveur d'impression 64
5.2 Résultat du monitoring d'un *PrintServer* 69

Listings

1.1 Exemple de déclaration IDL . . . 19
4.1 Structure du RequestInformations . . . 51
4.2 Structure du ReplyInformations . . . 52
4.3 Interface IDL de l'accesseur . . . 53
4.4 Interface IDL du RequestMonitor . . . 53
4.5 Interface IDL du ReplyMonitor . . . 54
4.6 Interface IDL du dispatcheur . . . 55
5.1 Initialisation de l'intercepteur . . . 63
A.1 ORBInitializer_impl.cpp . . . 81
A.2 ORBInitializer_impl.h . . . 82
A.3 ServerRequestInterceptor_impl.cpp . . . 83
A.4 ServerRequestInterceptor_impl.h . . . 87
A.5 Accessor.idl . . . 89
A.6 Accessor_impl.cpp . . . 89
A.7 Accessor_impl.h . . . 89
A.8 Dispatcher.idl . . . 91
A.9 Dispatcher_impl.cpp . . . 91
A.10 Dispatcher_impl.h . . . 96
A.11 Dispatcher_main.cpp . . . 97
A.12 Monitor.idl . . . 99
B.1 PrintServer.idl . . . 101
B.2 PrintServer_impl.cpp . . . 101
B.3 PrintServer_impl.h . . . 102
B.4 PrintServer_main.cpp . . . 102
B.5 PrintServerMonitor.cpp . . . 104
B.6 PrintServerMonitor.h . . . 108

Introduction

Même s'il ne s'agit pas d'un concept récent, l'utilisation de systèmes distribués est en constante croissance.

Cette évolution est principalement liée à l'augmentation des exigences en performance des applications informatiques. Une solution pour certaines d'entre elles est de les découper en plusieurs composants situés sur des machines différentes et qui communiquent entre-eux par l'intermédiaire d'un réseau informatique.

Pour réaliser cette coopération entre les composants, le développeur de l'application devra soit concevoir un protocole qui lui est propre, soit faire appel à un framework tel que CORBA. Une présentation de ce dernier fait l'objet du premier chapitre de ce mémoire.

Quelle que soit l'approche choisie, le développement présente un degré de difficulté supérieur si les applications sont de type distribué. Cette complexité accrue se traduit par une fréquence plus élevée d'erreurs tant lors de la conception que lors de l'implémentation de ces applications. La répartition des composants sur un ensemble de machines rend également plus difficile la détection et la localisation de ces erreurs.

Le monitoring, présenté dans le second chapitre, aide à résoudre ces problèmes en écoutant les conversations d'une application durant son exécution. Les informations ainsi récoltées peuvent ensuite être utilisées à des fins diverses, allant de la vérification de spécifications au contrôle d'autres applications.

Le troisième chapitre de ce mémoire est consacré à la description de quelques applications de monitoring utilisant CORBA. Cependant leurs fonctionnalités sont limitées à l'observation d'objets, elles ne permettent pas d'interagir avec eux. Nous avons estimé qu'il était important pour une telle application de pouvoir également avoir un rôle actif.

A cet effet nous vous présenterons dans le quatrième chapitre une architecture de monitoring distribué qui permet d'intercepter de façon transparente toutes les communications relatives à un objet CORBA. Celles-ci sont ensuite redistribuées à un ensemble de moniteurs indépendants s'ils en ont fait la demande.

Nous avons réalisé un prototype qui implémente cette architecture. Le cinquième chapitre explique comment l'intégrer dans une application existante et détaille l'implémentation d'un moniteur pour ce prototype au travers d'un exemple concret.

Une évaluation de l'architecture proposée fera l'objet du dernier chapitre. Nous y proposerons également des modifications qui permettraient d'améliorer les performances cette architecture et d'en étendre les fonctionnalités.

Chapitre 1

CORBA

Le *Common Object Request Broker Architecture* (CORBA) est un ensemble de spécifications développées par l'*Object Management Group* (OMG)[1] qui visent à définir un framework de développement d'objets distribués dans un milieu hétérogène.

L'OMG fût créé en 1989 par quelques compagnies à la tête d'un mouvement novateur en informatique. Actuellement, ce sont plus de 850 membres (constructeurs, développeurs, utilisateurs et universités) qui travaillent à faire évoluer ces standards.

Ce chapitre n'a pas la prétention d'être un exposé exhaustif de CORBA. Il a plutôt pour objectif d'énumérer et de présenter de manière succincte les concepts et les mécanismes qui sont utilisés par l'architecture de monitoring distribué développée dans le cadre de ce mémoire[2].

1.1 Les objets CORBA

L'OMG définit de manière abstraite un objet CORBA comme étant "*une entité identifiable qui fournit un ou plusieurs services qui peuvent être invoqués par un client*" [19]. Il s'agit d'une conception client/serveur classique dans laquelle l'objet qui implémente les services est un serveur tandis que l'application qui y fait appel est un client. Il est néanmoins tout à fait possible pour un objet de se comporter simultanément comme un client et un serveur. En plus des services qu'ils fournissent, les objets CORBA peuvent également disposer d'attributs.

Cette définition se rapproche fortement de la notion de classe pour les langages orientés objet, à la différence que les invocations peuvent se faire à distance. Tout comme une classe peut hériter d'une autre classe, un objet peut hériter d'un ou de plusieurs autres objets.

---

[1] http ://www.omg.org

[2] Pour de plus amples informations, voir [1, 6, 19].

1.1.1 La description des objets

Pour pouvoir décrire ses objets, l'OMG a mis au point un nouveau langage déclaratif nommé *Interface Description Language* (IDL). C'est ce langage qui permet de définir l'interface qu'implémente un objet, à savoir quels sont ses attributs, ses méthodes ainsi que ses relations d'héritage avec d'autres objets.

Types

Dans le langage IDL, tout attribut, paramètre de méthode et valeur de retour doivent obligatoirement être typés. L'IDL supporte les types simples (`long`, `float`, `string`,...), les types composés (structures, unions,...) et les références d'objets.

Invocations synchrones

Un premier mode d'invocation de fonction supporté par CORBA est l'appel synchrone. Il s'agit d'un appel qui suspend l'exécution du client tant que l'opération ne s'est pas terminée du côté serveur.

Ces appels peuvent être représentés en IDL avec les propriétés suivantes.

- Les paramètres de la fonction peuvent être accessibles soit en lecture (`in`), soit en écriture (`out`) ou une combinaison des deux (`inout`).
- A chaque fonction est associé un type de valeur de retour. Si celle-ci n'existe pas, ce type est appelé `void`
- L'exécution d'une fonction peut soulever une exception, soit au niveau de l'architecture CORBA comme lors de l'utilisation d'une référence invalide, soit au niveau de l'objet en lui-même pour signaler la survenance d'une erreur. Une exception se traduit par l'interruption de l'exécution de la méthode et l'envoi d'un message au client.

Remarquons également que dans CORBA tous les accès aux attributs sont considérés comme étant des invocations synchrones, et qu'ils ont donc pour conséquence d'interrompre l'exécution de l'objet qui effectue l'accès.

Invocations asynchrones

CORBA supporte également les invocations de méthodes asynchrones qui n'interrompent pas l'exécution du programme appelant. Il en résulte que ces requêtes sont plus limitées que les requêtes synchrones, en effet :

- il est impossible d'obtenir des valeurs de retour pour ces appels.
- les paramètres ne peuvent être que de type `in`, c'est à dire en lecture seule.
- ces invocations ne permettent pas la mise en place d'un mécanisme d'exceptions.

Listing 1.1 – Exemple de déclaration IDL

```
interface PrintSession
{
  oneway void reset();
  boolean print(in string data);
  void disconnect();
};

interface PrintServer
{
  boolean addUser(in string login, in string password);
  PrintSession connect(in string login, in string password);
};
```

- il n'y a aucune garantie que le serveur reçoive la requête émise.

Au niveau des déclarations IDL, ces méthodes asynchrones se distinguent par le mot-clef `oneway` qui se trouve devant la valeur de retour qui doit obligatoirement être `void`.

1.1.2 Exemple de description IDL

Le listing 1.1 qui représente un serveur d'impression basique permet d'illustrer la déclaration d'un serveur en IDL. Dans celui-ci sont déclarées deux interfaces : *PrintServer* et *PrintSession*.

L'interface *PrintServer* contient deux méthodes :

- `addUser` est la méthode qui permet d'ajouter des utilisateurs à ce serveur. Elle prend comme arguments un *login* et un mot de passe. Le résultat de l'opération est un booléen indiquant le succès de celle-ci.
- `connect` permet à un utilisateur qui est enregistré auprès du serveur d'obtenir un objet de type *PrintSession*.

Une fois créé, cet objet permet aux utilisateurs d'effectuer les opérations suivantes sur l'imprimante :

- `print` est la commande utilisée pour envoyer des données à l'imprimante. Tout comme l'ajout d'un utilisateur, le résultat de la méthode indique son succès.
- `reset` offre la possibilité d'interrompre l'impression des données qui ont été envoyées au moyen d'un `print`.
- `disconnect` sert à détruire la session d'impression lorsque celle-ci n'est plus utilisée.

1.1.3 L'interface statique

Le langage IDL permet de décrire les interfaces des objets indépendamment de tout langage de programmation, mais il ne permet pas d'en réaliser l'implémentation. Celle-ci peut se faire dans un langage au choix, pour autant que l'OMG ait défini des règles de *mapping* entre IDL et ce langage. Actuellement, ces règles sont définies pour la majorité des langages de programmation, ce qui permet d'implémenter et de faire communiquer entre eux des objets CORBA en Ada, Java, C, C++, Lisp, PL/1, COBOL, Python et SmallTalk.

Lorsque le choix du langage est établi, la description IDL de l'objet est projetée dans ce langage selon les règles de l'OMG au moyen d'un compilateur IDL. Les résultats de cette compilation sont le *stub* et le *skeleton* qui forment ensemble l'interface statique de cet objet ou *Statical Invocation Interface* (SII).

Le *stub* est un relais qui permet aux clients de s'adresser à un objet distant comme si celui-ci était local. De plus il permet de vérifier l'utilisation correcte des traductions de l'IDL en langage cible lors de la compilation des clients.

Le *skeleton* est quant à lui utilisé du côté serveur pour permettre la création d'un objet CORBA à partir de son implémentation, c'est-à-dire faire correspondre cette implémentation à la description de l'objet.

1.1.4 L'interface dynamique

CORBA dépasse cet aspect statique et permet aux objets de construire des requêtes de manière dynamique en utilisant un mécanisme défini sous le nom de *Dynamic Invocation Interface* (DII). L'avantage de cette interface dynamique en comparaison au *Static Invocation Interface* est qu'un programme qui désire invoquer des méthodes sur un objet distant ne doit pas obligatoirement disposer d'un stub généré à partir d'une description IDL de l'objet.

Du point de vue du serveur, il existe également un mécanisme similaire appelé *Dynamical Skeleton Interface* (DSI). Celui-ci lui permet de répondre à des requêtes sans pour autant que celles-ci aient été préalablement définies dans son interface IDL. Il en dérive que le DSI ne puisse être utilisé que conjointement au DII.

Application

Le *débuggage* d'applications distribuées bénéficie largement de cette autre interface. Si on désire avoir accès aux attributs et aux méthodes d'un objet au moyen du SSI nous devons disposer, lors du développement du *débuggeur*, des descriptions IDL des interfaces afin de pouvoir générer les *stubs*.

Si on utilise le DII, la disponibilité de ces *stubs* n'est plus indispensable. Tout ce qui est nécessaire, c'est de disposer d'un `Interface Repository` (IR) tel qu'il est défini

par l'OMG. Cet IR permet d'accéder à la description d'une interface à partir de son nom.

1.1.5 Les références d'objets

Lors de sa création, à chaque objet CORBA est attribuée une référence unique appelée *Interoperable Object Reference* (IOR). Celle-ci contient des informations de localisation et d'identification qui le concerne, ce qui permet aux développeurs de le manipuler sans contraintes de localisation.

Afin de standardiser l'IOR, l'OMG a fixé les informations que celle-ci doit contenir. Il s'agit :

- du nom complet de l'interface IDL de l'objet
- des renseignements qui permettent de contacter le serveur. A titre d'exemple, pour un bus CORBA qui utilise le protocole TCP/IP pour les communications entre objets, il s'agit d'une adresse IP ainsi que d'un port
- un identificateur unique de l'objet au niveau du serveur

A ces informations indispensables, des services peuvent adjoindre des informations complémentaires afin d'étendre l'IOR. Cette fonctionnalité du bus CORBA sera abordée à la section 1.4.

1.2 Le bus CORBA

Le bus CORBA, autrement connu sous le nom d'*Object Request Broker* (ORB), est l'ensemble des composants qui permettent à des objets de communiquer entre eux, en masquant les différences relatives à la localisation ainsi que celles liées à l'hétérogénéité de l'environnement comme le langage, l'architecture matérielle et l'implémentation de CORBA utilisée.

Une implémentation classique d'un bus CORBA contient donc les *stubs* et *skeletons* générés à partir des descriptions IDL des objets. Elle contient aussi généralement une librairie qui fournit les fonctionnalités de base du bus, par exemple la localisation d'objets et la transmission des requêtes.

Depuis la version 2.0 de CORBA, il est possible d'interconnecter les bus CORBA au moyen du *General Inter-ORB Protocol* (GIOP), qui définit :

- une représentation standardisée des données, *Common Data Representation* (CDR) qui permet d'échanger des informations entre des architectures matérielles différentes
- les références d'objets IOR (voir 1.1.5)
- les messages relatifs aux requêtes, aux réponses, aux exceptions, ...

Ce protocole est, comme son nom l'indique, indépendant du moyen de transport physique. Une concrétisation de ce protocole est l'*Internet Inter-ORB Protocol* ou IIOP qui utilise TCP/IP comme couche de transport.

1.3 Localisation des composants

Nous allons aborder la manière dont les clients peuvent obtenir les références des objets CORBA.

Les références des objets CORBA peuvent être utilisées soit comme paramètre ou soit comme résultat d'une méthode. De cette manière, si un serveur crée un ou plusieurs objets, il suffit de retourner leurs références comme résultat de l'opération. Cependant, cette procédure ne résout pas le problème de la localisation du serveur initial.

Une solution proposée pour contourner cette difficulté est de transformer les IOR en chaînes de caractères selon des conventions édictées par l'OMG. Ce sont les *stringified IOR*. Ceux-ci peuvent être sauvés par le serveur dans un fichier et récupérés plus tard par le client pour être retransformés en références. Cette technique peu élaborée, ne s'applique pas à un système distribué de grande échelle.

Pour répondre à ces besoins l'OMG a crée le *Naming Service*. Il s'agit d'un objet CORBA qui permet d'associer des chaînes de caractères à des références d'objets. Il peut donc être comparé à un annuaire téléphonique. Évidemment, comme le *Naming Service* est un objet comme un autre, il faut au préalable avoir obtenu son IOR. Ceci peut être réalisé en recourant une seule fois aux *stringified IOR*. Dès que cette opération est effectuée, tous les objets qui y sont inscrits deviennent accessibles.

Pour compléter cet annuaire il existe également un service nommé le *Trader Service* qui permet de trouver un objet non pas à partir d'une chaîne de caractères, mais à partir de ses propriétés.

1.4 Les intercepteurs CORBA

Jusqu'il y a peu, il n'était pas possible de rajouter des services à l'ORB si ce n'est en modifiant directement son implémentation. Cette façon de procéder est non seulement fastidieuse, mais en plus elle limite le développeur à une implémentation de CORBA particulière.

Les intercepteurs CORBA offrent désormais la possibilité d'intégrer des services qui peuvent modifier le déroulement des requêtes ainsi que transmettre de manière portable, des informations relatives à un service (appelés contextes de service ou *service contexts*) dans l'ORB.

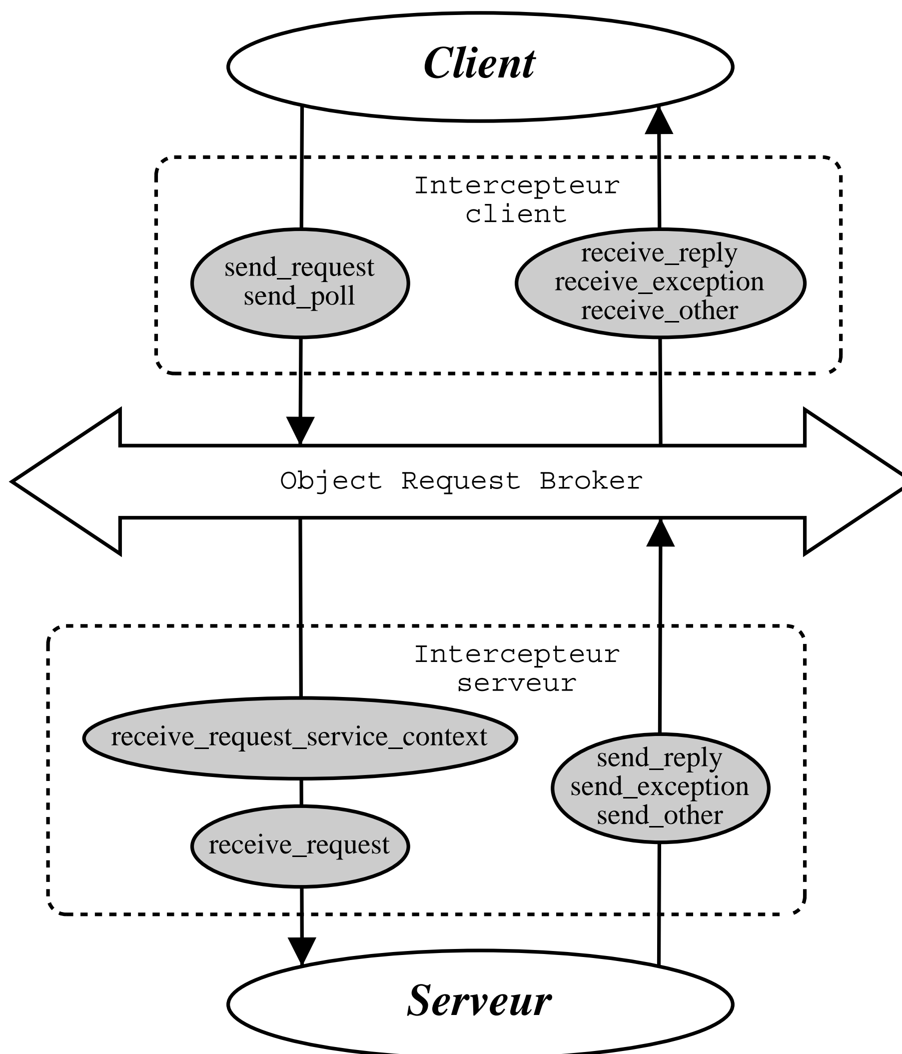

FIG. 1.1 – Points d'interception

Depuis CORBA 2.3, cette *portabilité* est garantie par des spécifications précises et par une description de leur interface en pseudo IDL. Les intercepteurs sont donc des objets CORBA qui ont cependant la particularité d'être locaux, c'est à dire non-accessibles pour des objets distants.

Il existe deux catégories d'intercepteurs : les intercepteurs de requêtes (*Request Interceptors*) et les intercepteurs d'IOR[3] (*IOR Interceptors*).

Les intercepteurs d'IOR permettent à des services de rajouter des informations à la référence d'un objet lorsque celui-ci est créé. Nous ne rentrerons pas dans les détails de leur utilisation car ils ne sont d'aucun intérêt dans le cadre de ce mémoire.

Les `Request Interceptors` sont plus intéressants, ils permettent aux services de l'ORB de transférer des informations entre le client et le serveur ainsi que d'accéder

[3]Interoperable Object Reference

aux informations qui composent leurs communications.

Avant de procéder à la description des différents points d'interception existants, nous soulignons qu'il est possible d'avoir plusieurs intercepteurs enregistrés simultanément. La question de l'ordre d'activation de ceux-ci sera traitée au point 1.4.3.

1.4.1 Intercepteurs au niveau du client

Il existe cinq points d'interception distincts dans l'intercepteur client. Le `send_request` et le `send_poll` permettent d'intercepter les requêtes émises par un objet avant que celles-ci ne pénètrent l'ORB, le `receive_reply`, le `receive_exception` et le `receive_other` s'occupent des réponses qui sont renvoyées au client. Ces différents points sont repris à la figure 1.1.

Il est possible pour un objet CORBA d'interroger un serveur pour connaître le résultat d'une requête quelque soit l'objet à l'origine de l'invocation. Au niveau du client, les requêtes de ce type sont interceptées par le point `send_poll`.

Le second point d'interception situé entre le client et l'ORB est le `send_request`. Il offre la possibilité à un intercepteur de consulter les informations qui concernent la requête et de modifier les contextes de service avant qu'ils n'entrent dans l'ORB.

Dans le cas d'une opération réussie, la réponse est interceptée par le point `receive_reply` avant que le résultat ne soit communiqué au client et l'intercepteur a accès aux informations qui constituent cette réponse.

Une première situation alternative résulte de l'émission d'une exception quelque part sur le chemin de l'opération ce qui provoque l'appel de la méthode `receive_exception` de l'intercepteur. Les exceptions peuvent être émises en tous points d'interception (client et serveur) ainsi que par l'objet cible.

Il est aussi possible pour un intercepteur de soulever une exception, ce qui a pour conséquence que les autres intercepteurs n'auront pas leur méthode `send_request` invoquée. En revanche, selon le type d'exception, les points d'interception suivants sont activés :

- Dans le cas d'un `ForwardRequest`, c'est à dire une exception qui provoque la ré-émission de la requête avec comme destination l'objet initial ou un autre objet, les autres intercepteurs reçoivent un `receive_other`.
- Dans le cas contraire, appelé exception système, les intercepteurs reçoivent un `receive_exception`.

1.4.2 Intercepteurs au niveau du serveur

Du côté du serveur, il existe également cinq points d'interception. Avant qu'une requête n'atteigne sa destination celle-ci passe d'abord par le

	receive_request_service_contexts	receive_request	send_reply	send_exception	send_other
Request_id	Oui	Oui	Oui	Oui	Oui
Operation	Oui	Oui	Oui	Oui	Oui
Arguments	Non	Oui	Oui	Non	Non
Exceptions	Non	Oui	Oui	Oui	Oui
Contexts	Non	Oui	Oui	Oui	Oui
Result	Non	Non	Oui	Non	Non
Response_expected	Oui	Oui	Oui	Oui	Oui
Reply_status	Non	Non	Oui	Oui	Oui
Forward_reference	Non	Non	Non	Non	Oui
Object_id	Non	Oui	Oui	Oui	Oui
Adapter_id	Non	Oui	Oui	Oui	Oui
Target_most_derived_interface	Non	Oui	Non	Non	Non

TAB. 1.2 – Informations disponibles pour les points d'interception serveur

`receive_request_service_contexts` et ensuite par le `receive_request`. Selon le résultat de l'opération, la réponse sera interceptée par un des points d'interception suivants : le `send_reply`, le `send_exception` ou le `send_other`.

Le premier point, c'est-à-dire le `receive_request_service_contexts`, a pour objectif de permettre aux intercepteurs de récupérer leurs contextes de service qui auraient pu être rajoutés à la requête par le `send_poll` ou par le `send_request` d'un intercepteur du client.

Certaines informations concernant la requête ne sont pas encore disponibles à ce point, il s'agit par exemple des arguments des opérations. Il est également impossible d'obtenir certaines informations concernant l'objet cible et notamment son interface, son identificateur ainsi que celui de l'adaptateur d'objet qui le relie à l'ORB. Le tableau 1.2 reprend les différents renseignements disponibles en ce point.

Contrairement au point précédent, le `receive_request` donne accès à l'entièreté des informations qui composent la requête, exceptées évidemment les informations qui concernent le résultat de son exécution.

S'il n'y a pas eu d'exceptions à ces deux points d'entrée, la requête atteint l'objet où elle provoque l'exécution de l'opération demandée.

Lorsque nous avons présenté les intercepteurs au niveau des clients, nous avons vu que le résultat d'une requête pouvait être une réponse, une exception ou un message

qui n'est ni une réponse normale ni une exception. Cette classification des réponses se retrouve également du côté du serveur sous la forme de trois points d'interception le `send_reply`, le `send_exception` et le `send_other`.

Le `send_reply` correspond ainsi au `receive_reply` du client, il est appelé lorsque le serveur a terminé avec succès l'exécution de l'opération demandée. C'est à cet endroit qu'un intercepteur peut prendre connaissance des résultats avant que ceux-ci ne soient transmis au client.

Le point `send_exception` est invoqué lorsque le serveur renvoie une exception au client, ce qui permet à l'intercepteur de consulter les informations concernant l'exception et de modifier le contexte de service avant que l'exception n'atteigne le client.

Le point `send_other` correspond à une réponse qui n'est ni le résultat d'une requête réussie, ni une exception. Le plus souvent il s'agira d'un résultat de *ForwardRequest*, c'est-à-dire une demande de retransmission de la requête, sur le même objet ou sur un autre.

Tout comme le `send_request` et le `send_poll` des intercepteurs du client, ces points d'interception peuvent soulever une exception système ou un `ForwardRequest`.

- Dans le cas d'une exception système, les autres intercepteurs reçoivent un `send_exception`.
- Dans le cas d'un `ForwardRequest` ce sont les points d'interception `send_other` des autres intercepteurs qui sont activés.

1.4.3 Ordre d'activation des intercepteurs

Comme évoqué précédemment au point 1.4, plusieurs intercepteurs peuvent être enregistrés simultanément. Nous avions laissé de côté la question de l'ordre de leur activation.

La règle générale est que dans le sens du client vers le serveur les intercepteurs soient appelés dans l'ordre où ils ont été enregistrés auprès de l'ORB. Lorsqu'il s'agit d'une réponse venant du serveur, ils sont appelés dans le sens inverse de leur inscription.

Prenons par exemple le modèle de la figure 1.2 pour lequel trois intercepteurs client ont été enregistrés dans l'ordre suivant : `A` suivi de `B`, suivi de `C`.

Si une requête est émise, l'ordre d'activation des points `send_request` sera `A`→`B`→`C`. Lorsque la réponse reviendra du serveur, c'est dans l'ordre `C`→`B`→`A` que les intercepteurs seront activés.

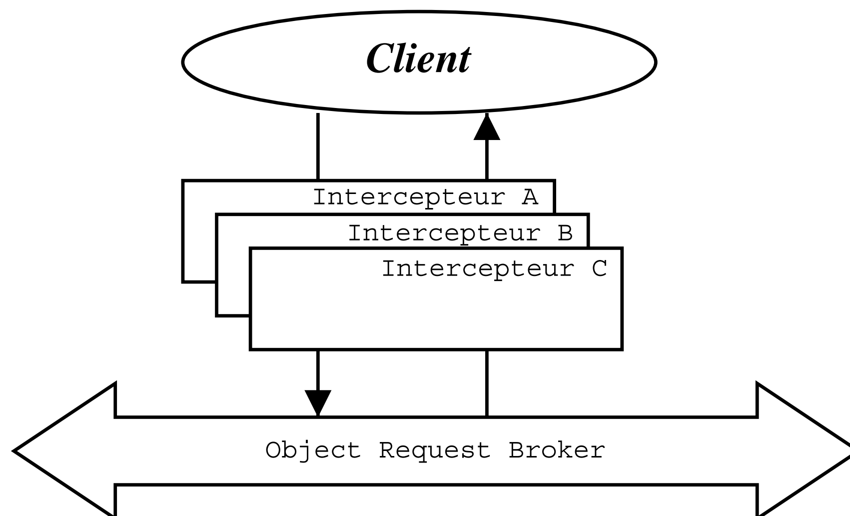

Fig. 1.2 – Ordre d'activation pour les intercepteurs du client

1.4.4 Exemples d'applications des intercepteurs

Implémentation d'une fonction d'authentification

Pour illustrer la mise en place d'un tel service, nous prendrons l'exemple d'un serveur pour lequel on voudrait une authentification des clients, dans le but d'empêcher une personne non-autorisée d'effectuer des opérations.

Une première manière d'implémenter cette fonctionnalité est celle d'inclure des fonctions d'authentification au sein de l'objet. Cette approche a ses limites, car même s'il est possible d'ajouter aux requêtes des paramètres qui permettent de gérer l'authentification du client il n'est pas possible d'empêcher l'accès à ses attributs.

Une approche utilisant les intercepteurs serait celle d'adjoindre les informations d'authentification du client à la requête lorsque celle-ci passe par le point `send_request` et de récupérer celles-ci dans l'intercepteur du serveur au point `receive_request`. Si les données sont validées, alors l'invocation suit son cours. Dans le cas d'une authentification erronée, une exception est renvoyée au client ce qui a pour conséquence de l'empêcher d'accéder à l'objet.

Cette application n'est présentée qu'à titre d'exemple car elle permet de comprendre de manière simplifiée le fonctionnement du processus d'authentification au moyen des intercepteurs. Nous décrirons maintenant un service mis au point par l'OMG, dont les fonctionnalités sont bien plus étendues.

Le service de sécurité

Le service de sécurité qui a été défini par l'OMG [18] vient se greffer sur l'ORB et permet de gérer les différentes fonctions suivantes :

- l'**identification** des acteurs, que ce soit des humains ou des objets, afin de vérifier s'ils sont réellement ceux qu'ils prétendent être.
- l'**autorisation** d'un acteur à effectuer certaines opérations. Pour cela, il se base sur l'identité de l'acteur ainsi que sur les attributs de contrôle qui sont associés aux objets sécurisés.
- l'**audit de sécurité** qui permet même lors d'une chaîne d'invocations, d'associer une opération à la personne qui est à son origine..
- la **sécurité des transactions**, c'est à dire la vérification de l'intégrité des messages qui sont échangés entre les objets, ainsi que de manière optionnelle une *encryption* de ceux-ci.
- la **non-répudiation** de la part des acteurs qui ne peuvent pas prétendre ne pas être à l'origine d'un message qui a été émis sous leur identité, voire nier la réception d'un message qui leur aurait été transmis correctement.
- L'**administration** des différentes informations, notamment les politiques de sécurité de l'application.

Outre sa richesse au niveau des fonctionnalités proposées, comme nous l'avons souligné, l'avantage d'utiliser ce service est qu'il est défini par l'OMG. Cette définition officielle a pour conséquence qu'une application qui en fait usage pourra être adaptée à un autre ORB sans trop de modifications.

Le service de transaction

Le concept de transaction est important pour pouvoir développer des applications fiables, notamment lorsqu'elles contiennent des accès concurrents à des ressources partagées. Utilisé à l'origine pour les bases de données, ce concept s'est étendu aux applications distribuées.

Une transaction est définie comme une unité de travail qui respecte des propriétés communément appelées `ACID`, autrement dit :

- **Atomicité** : si une erreur survient dans le courant d'une transaction, toutes les modifications qui ont été effectuées dans celle-ci sont annulées. Cette opération est appelée *rollback*. Cela signifie que soit toutes, soit aucune opération d'une transaction sont réalisées.
- **Consistance** : cette propriété assure qu'une transaction fait passer l'objet d'un état consistant à un autre état qui l'est également. Autrement dit, les transformations apportées par une transaction préservent certains invariants.
- **Indépendance** : l'exécution concurrente de transactions ne provoque pas d'interférences, c'est à dire que les modifications effectuées dans une transaction

ne sont pas visibles pour des objets extérieurs tant que celles-ci n'ont pas été validées au moyen d'un *commit*.

- **Durabilité** : lorsqu'une transaction est validée, les changements sont visibles depuis l'extérieur de l'objet.

La fonction des intercepteurs dans ce contexte est d'assurer la coopération entre le service de transactions et l'ORB. En effet, des vérifications doivent être effectuées du côté client avant qu'une requête ne soit envoyée vers un objet cible, et du côté serveur lorsqu'une requête est reçue. Pour de plus amples informations, nous vous proposons de consulter les documents officiels de l'OMG [17].

Chapitre 2

Monitoring

Nous pouvons définir le monitoring comme l'activité qui consiste à récolter des informations lors de l'exécution d'un programme. Celles-ci sont utilisées pour vérifier ou visualiser le fonctionnement de l'application mais aussi pour modifier le comportement du système en fonction des résultats observés. Nous présenterons dans dans la section 2.1 de ce chapitre quelques applications dans le cadre des systèmes distribués du monitoring.

La plupart des fonctions qui seront énumérées pourraient être directement intégrées au programme. Néanmoins, la mise en place d'un monitoring externe présente plusieurs avantages.

- La séparation entre l'implémentation et la vérification permet de scinder le business et le contrôle. Il est de plus possible d'ajuster dynamiquement la fonction de vérification selon le contexte d'utilisation (Internet vs Intranet).
- Le moniteur peut être adjoint au programme après la conception du logiciel.
- Le langage d'implémentation des moniteurs est indépendant de celui de l'application, ce qui permet l'utilisation de langages spécialisés, comme par exemple des langages à base de règles.
- Par ailleurs, la mise en place d'un moniteur externe, permet de décharger l'application de tâches susceptibles de ralentir sa vitesse d'exécution.

Dans le cadre de systèmes distribués, le monitoring génère des difficultés qui ne sont pas rencontrées dans le cas des applications non-distribuées. Celles-ci seront présentées dans la section 2.2.

La dernière partie de ce chapitre sera consacrée aux différentes techniques qui permettent d'obtenir les informations nécessaires au monitoring.

2.1 Les fonctions du monitoring

2.1.1 Vérification de la conformité aux spécifications

Le monitoring a comme fonction essentielle de vérifier la conformité des éléments interceptés. Pour se faire, il se base sur les deux types de spécifications inscrites au cahier des charges, à savoir les spécifications fonctionnelles et non-fonctionnelles.

Spécifications fonctionnelles

Les spécifications fonctionnelles des différents composants représentent la première étape de la conception d'une application distribuée. Elles couvrent le détail des opérations de ces composants ainsi que leurs interactions.

Celles-ci sont ensuite implémentées et donnent naissance à un programme. Il est cependant fréquent que des erreurs de programmation ou une mauvaise compréhension des spécifications soient à l'origine de comportements erronés. Il est donc nécessaire, pendant son exécution, de pouvoir vérifier si les spécifications établies sont bien respectées.

Habituellement, cette fonction est effectuée d'une part grâce à l'insertion d'opérations qui vérifient la validité des paramètres et d'autre part au moyen de l'affichage d'informations lors de l'exécution du programme. Ces deux procédés permettent la détection et la localisation de tout comportement anormal.

Bien que cette approche soit fortement répandue, elle comporte néanmoins deux limitations qui la rendent difficilement utilisable dans le cadre du développement d'applications distribuées.

- En premier lieu, les informations récoltées ne sont habituellement pas centralisées, ce qui implique la nécessité de devoir faire un recoupement de données afin d'avoir une vue d'ensemble du système.
- En second lieu, la vérification est réalisée au niveau du composant ce qui peut avoir des conséquences non négligeables sur les performances du système.

La surveillance par un monitoring externe fait disparaître ces deux inconvénients.

Indépendant de l'application, le monitoring analyse les paramètres et les résultats des opérations. Il détermine s'ils satisfont aux propriétés initialement définies. Ces propriétés sont également appelées préconditions et postconditions.

Observateur des messages échangés entre les différents composants, il permet également de vérifier l'enchaînement correct et la logique des différentes opérations.

Si nous reprenons l'exemple présenté dans le listing 1.1, une précondition serait d'imposer l'utilisation d'un mot de passe comportant un nombre minimum de caractères pour la méthode `addUser` du serveur d'impression. Il est de plus possible de véri-

fier l'ordre d'invocation des méthodes d'une session afin de s'assurer que la fonction `reset` est bien invoquée pendant l'exécution d'un `print`.

Lorsque le composant qui assure la fonction de monitoring constate une erreur, il peut réagir en effectuant un enregistrement destiné à une consultation ultérieure, ou en prenant l'initiative de faire avorter l'opération.

Spécifications non-fonctionnelles

Les spécifications non fonctionnelles visent l'environnement d'exécution de l'application. Celles-ci peuvent, tout comme les spécifications fonctionnelles, être vérifiées au moyen de l'analyse de l'observation des communications entre les composants.

C'est ainsi qu'en mesurant le délai qui sépare la réception d'une requête et l'envoi de la réponse, il est possible de déterminer la durée de réponse d'une opération. Celle-ci est ensuite comparée aux exigences des spécifications.

Le monitoring assure également le contrôle et la sécurisation d'une application. Il est en effet fréquent d'être en présence de composants dont l'accès doit être réservé et sécurisé. L'observation des messages destinés à ces composants permet d'en contrôler le contenu et les propriétés, comme par exemple les adresses sur le réseau du client, afin de déterminer si l'accès est réellement autorisé.

2.1.2 Contrôle

Nous avons vu dans le chapitre précédent que CORBA cache aux programmeurs les détails qui concernent les communications entre objets afin de faciliter le développement de systèmes distribués.

Cette abstraction comporte néanmoins une limitation : il n'est généralement pas possible pour une application de changer dynamiquement la configuration de l'ORB. Pour combler cette lacune, une nouvelle génération de *middleware*s est en train de voir le jour : il s'agit des *reflective middleware*s [8]. Ceux-ci peuvent, contrairement à leurs prédécesseurs, être modifiés pour s'adapter à l'environnement dans lequel ils sont utilisés.

Dans ce contexte d'environnement changeant, il est possible d'attribuer un rôle de contrôle aux moniteurs : ce sont ceux-ci qui vont détecter tout changement dans les conditions d'exécution d'un programme et qui vont configurer l'ORB afin que ce dernier puisse s'y adapter.

2.2 Difficultés liées au monitoring de systèmes distribués

Le monitoring d'applications distribuées soulève des difficultés liées aux communications qui sont effectuées entre les objets. En effet, la transmission de messages à travers un réseau introduit un certain nombre de complications qui ne surviennent pas lorsque les applications résident sur une seule machine.

Les délais dans le transfert des informations

L'utilisation de réseaux informatiques pour véhiculer des informations introduit inéluctablement des délais entre l'émission et la réception d'un message. Ceux-ci sont responsables du fait que les données envoyées peuvent ne plus être d'actualité à l'instant où elles sont reçues par le moniteur.

Cette affirmation concerne exclusivement les systèmes qui envoient les informations de monitoring de manière asynchrone. Dans le cas adverse de messages synchrones, l'application est interrompue pendant la transmission. Cette interruption n'invalide pas les données, mais elle peut avoir un impact négatif important au niveau des performances de l'application monitorée.

Les déséquencements des messages

Lors de l'envoi d'informations de monitoring de manière asynchrone, la variabilité des délais peut provoquer un déséquencement des messages. Ce déséquencement a pour conséquence de rendre difficile l'établissement de relations de cause à effet entre ceux-ci.

L'absence de temps universel

Les délais ainsi que leur variabilité rendent également impossible la synchronisation des horloges sur les différentes machines qui composent le système. Bien qu'il existe des protocoles tel que NTP (*Network Time Protocol*, [10]) ou SNTP (*Simple Network Time Protocol*, [11]) qui permettent de synchroniser l'horloge d'une machine avec un serveur de référence, ceux-ci parviennent tout au plus à réduire l'erreur et à limiter la dérive du temps.

La saturation du moniteur par le nombre d'objets générés

La bande passante de la connexion inter-objets représente également une contrainte relative aux réseaux informatiques. Lorsqu'un grand nombre d'objets essaient de

transmettre simultanément des informations au moniteur, cet afflux important et concomitant peut provoquer une congestion du réseau.

La nécessité d'une convention de représentation commune des données

L'hétérogénéité des systèmes distribués nécessite l'établissement d'une représentation commune des données. Pour CORBA, celle-ci est définie par l'OMG sous le nom de *Common Data Representation*.

L'opération de conversion des données dans le format standardisé, aussi appelée *marshalling*, peut avoir une influence néfaste sur les performances du monitoring. Il en va de même pour l'opération inverse, le *demarshalling*, qui est effectuée par le récepteur afin de traduire les informations encodées.

2.3 Les mécanismes d'interception

Après avoir défini dans les sections précédentes les fonctions du monitoring, il nous semble nécessaire de passer en revue les mécanismes permettant d'obtenir les informations nécessaires au monitoring.

Ceux-ci se différencient par leur transparence, leur portabilité et leur interactivité. Nous allons également préciser leurs restrictions ainsi que leurs avantages.

2.3.1 Le sniffing sur le réseau

Un premier procédé relativement simple est l'interception des communications TCP/IP qui transitent sur le réseau et l'analyse des messages IIOP de CORBA pour en extraire les informations qui sont nécessaires. Ethereal[1] est un exemple d'outil qui permet d'analyser les traces réseau représentant des invocations de méthodes CORBA.

Cette méthode a l'avantage d'être transparente aussi bien pour les serveurs et les clients que pour l'ORB. Ses limites sont par contre importantes et réduisent son utilisation à des systèmes simples. Les restrictions engendrées par ces limites sont les suivantes :

- cette technique est principalement applicable lorsque le système communique au moyen d'un réseau qui fonctionne en mode *broadcast*. Dans le cas d'un réseau *switché* il faut impérativement sniffer sur les interfaces des différentes machines et synchroniser les résultats.

[1] http ://www.ethereal.com

- une autre difficulté liée à cette approche consiste dans l'impossibilité de modifier le contenu des messages ce qui rend impossible toute interactivité au niveau du monitoring.
- lorsque les objets sont localisés sur une même machine, certaines implémentations de CORBA court-circuitent le réseau en permettant aux objets de dialoguer entre eux par l'intermédiaire des communications inter-processus. Lorsque de telles optimisations sont mises en œuvre, l'utilisation de cette technique est rendue impossible.

2.3.2 Les intercepteurs au niveau de l'OS

Les intercepteurs au niveau de l'OS ont la capacité d'intercepter les messages relatifs aux communications entre objets. Ils procèdent en redéfinissant les fonctions de l'API réseau du système d'exploitation au moyen de librairies dynamiques. Ces fonctions ainsi redéfinies se placent entre l'ORB et la couche réseau du système d'exploitation ce qui leur permet d'intercepter les messages. Il est également possible de faire appel au système de fichiers */proc* présent dans les systèmes UNIX comme le fait l'outil appelé *Eternal* [13].

Cette procédure a l'avantage d'être indépendante de l'ORB utilisé et de fonctionner en totale transparence aussi bien pour les serveurs que pour les clients. Cependant, on retrouve les inconvénients énoncés précédemment pour le sniffing sur le réseau. D'une part l'interception des messages ne peut se faire que localement et d'autre part il est nécessaire de les *demarshaller* afin de pouvoir disposer des informations utiles.

Contrairement à la méthode de sniffing sur le réseau qui est dépendante de l'architecture du réseau, la méthode des intercepteurs au niveau de l'OS est particulièrement dépendante du système d'exploitation. Cette dépendance complique le mécanisme d'interception dès que l'on travaille dans un environnement hétérogène.

2.3.3 L'instrumentation des stubs et des skeletons

Dans le cas d'invocations qui n'utilisent pas d'interfaces dynamiques, on a la certitude que les messages passent par les stubs et les skeletons générés au moyen du compilateur IDL à partir des définitions des interfaces. Il est dès lors possible d'y intégrer des fonctionnalités de monitoring soit en modifiant le compilateur si les sources de celui-ci sont disponibles, soit en modifiant le code généré par ce dernier.

Deux difficultés majeures sont liées à cette manière de procéder.

- Le code généré par le compilateur IDL est spécifique à l'implémentation CORBA utilisée, ce qui limite la portabilité des modifications apportées.

- Lors d'invocations ou de traitement de requêtes dynamiques, les messages ne transitent pas par les stubs et les skeletons. Cette technique ne peut dès lors pas être utilisée pour intercepter les messages issus de requêtes dynamiques.

2.3.4 L'utilisation de proxy

Pour monitorer les appels de méthodes sur un serveur, il est aussi possible d'utiliser un objet qui possède une interface similaire à celle de l'objet que l'on désire monitorer, de manière telle que celui-ci vérifie l'exactitude de la requête et la retransmet ensuite vers le serveur [23].

Même si cette technique assure généralement une transparence de procédure pour le serveur, il n'en est pas toujours de même pour le client. En effet, si un serveur renvoie à un client la référence d'un objet, celle-ci n'est pas couverte par le proxy et les invocations sur cet objet ne seront pas monitorées.

2.3.5 Les intercepteurs CORBA

Les intercepteurs dont nous avons présenté le fonctionnement à la section 1.4, permettent d'intercepter tous les messages destinés à un objet. Ils ne présentent pas les inconvénients soulignés pour les méthodes précédentes.

- Les intercepteurs CORBA permettent d'intercepter les messages entre objets qu'ils soient distants ou localisés sur la même machine.
- Ils sont indépendants du système d'exploitation utilisé ainsi que de l'architecture physique du réseau.
- L'utilisation des intercepteurs est invisible pour les clients et les serveurs.
- Pour être mis en place, les intercepteurs CORBA ne nécessitent pas de modifications importantes au niveau de l'application qui doit être monitorée.

Chapitre 3

Étude de l'existant

Parmi la multitude d'infrastructures de monitoring existantes, nous en avons retenu trois qui approchent les objectifs que nous nous sommes fixés lors de la mise en place des lignes directrices de ce mémoire. Quelques unes ont abouti sous la forme d'une application, d'autres n'en sont restées qu'à la description d'une architecture générale.

3.1 CorbaTrace

Le projet CorbaTrace a été développé par cinq étudiants de l'Université de Nantes(France) dans le cadre de leur travail de fin d'études, un DESS en Génie Informatique. Il avait pour objectif de poursuivre le projet développé précédemment par deux étudiants qui ont effectué un *masters*.

Il a donné lieu à une application écrite en Java qui est disponible gratuitement sous la licence publique Library General Public Licence[1] (LGPL) à l'adresse suivante : `http ://corbatrace.tuxfamily.org`.

Objectif

L'ambition de cet outil est de permettre de mieux comprendre le fonctionnement d'une application distribuée en présentant de manière graphique les communications entre les composants de l'architecture. Cette analyse des interactions est effectuée à posteriori à partir de fichiers de *log* créés pour chaque objet lors de l'exécution de l'application.

[1] http ://www.gnu.org/licenses/lgpl.html

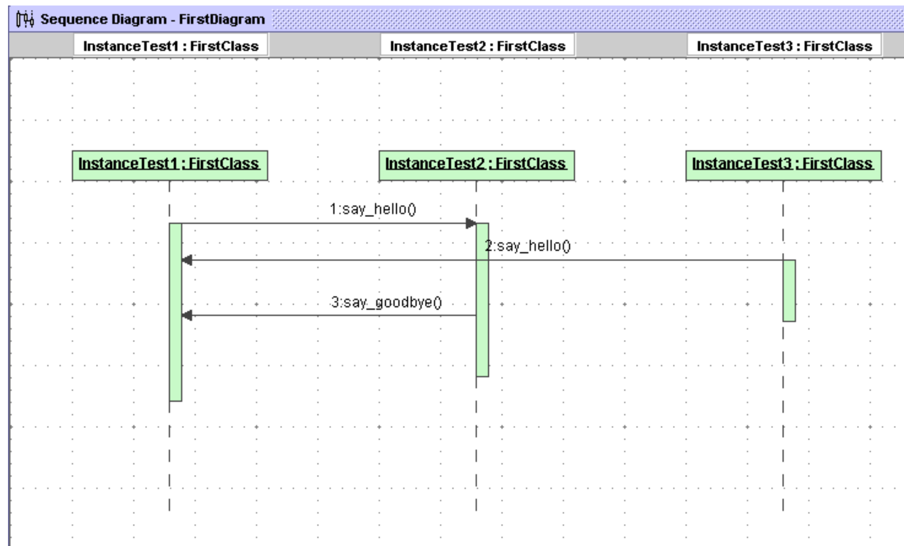

Fig. 3.1 – Monitoring : Visualisation d'un fichier XMI à l'aide de MagicDraw UML

Mise en œuvre

Dans ces fichiers sont enregistrés les différents événements relatifs à la réception des requêtes ainsi qu'à l'envoi de leurs réponses. Le format utilisé pour le stockage de ces informations est le XML. Celui-ci a l'avantage d'être facile à manipuler et d'être indépendant de l'architecture sur laquelle il a été créé.

Pour enregistrer les messages concernant l'activité d'un objet, il est fait usage d'intercepteurs CORBA portables tels qu'ils sont définis dans la version 2.3 des spécifications. CorbaTrace fait appel à ces intercepteurs au niveau des clients et des serveurs afin de disposer d'un maximum d'informations entre l'invocation et la réception de la réponse. CorbaTrace utilise également les contextes de services pour transmettre des données qui permettent la mise en correspondance des messages issus des clients et des serveurs.

Ces différents *logs* offrent la possibilité de visualiser l'activité d'un seul composant durant son exécution. Il n'en reste pas moins que pour obtenir une analyse de l'ensemble des interactions, il faut procéder à une intégration des logs individuels.

Cette démarche est réalisée au moyen d'un programme appelé `Log2XMI` qui élabore à partir de ces fichiers un diagramme de séquencement UML sauvegardé dans un fichier au format XMI[2] (XML Metadata Interchange).

La dernière opération est la visualisation des données au moyen d'une application externe telle que Rational Rose ou MagicDraw UML. Le résultat de cette opération est présenté à la figure 3.1.

[2] http ://xml.coverpages.org/xmi.html

3.2 MODOCC - MOnitoring of Distributed Object and Component Communication

MODOCC, précédemment connu sous le nom de *Monitoring*, est une application qui a été développée par quatre chercheurs Hollandais de l'Université de Twente au Pays-Bas. Ceux-ci, Nikolay K. Diakov, Harold J. Batterman, Hans Zandbelt et Marten J. van Sinderen sont les auteurs de différents articles qui traitent du monitoring des interactions entre les composants d'un système distribué [2, 3, 4].

Mise en œuvre

Contrairement à CorbaTrace, l'interception des requêtes et des réponses ne se fait pas au niveau des intercepteurs CORBA mais dans les stubs et les skeletons, comme expliqué au point 2.3.3. Ces derniers sont générés par un compilateur IDL modifié[3] et contiennent des instructions permettant de relayer les informations qui transitent par leur intermédiaire, notamment les invocations de requêtes.

Une requête émise par un client passe deux fois par le stub et deux fois par le skeleton, ce qui permet de disposer d'informations à quatre points :

- du côté client, à l'émission de la requête ou à la réception de la réponse
- du côté serveur, lorsque la requête est reçue ou que la réponse correspondante est renvoyée

Le composant vers lequel ces différents événements sont envoyés est appelé le *LocalMonitor*. Celui-ci a comme fonction de les redistribuer à des moniteurs spécialisés appelés les *CentralizedMonitor*. Ces sont ces derniers qui vont analyser, enregistrer et présenter les informations de monitoring.

Le contenu des événements qui sont transmis à ces *CentralizedMonitors* porte sur les points suivants :

- l'heure précise de l'émission du message ainsi qu'un identifiant de la requête qui permettent de mettre en relation les différents évènements.
- les informations qui identifient l'objet qui est à l'origine de l'événement.
- les caractéristiques de l'événement qui dépendent de la nature de celui-ci. Dans l'exemple d'une invocation de méthode, il s'agit du nom de l'interface, du nom de l'opération ainsi que ses arguments.

La problématique de la transmission d'un contexte qui accompagne une requête tout au long de son parcours a été résolue au moyen des intercepteurs.

[3] basé sur JacORB, voir http ://www.jacorb.org

Évaluation

Dans cette architecture, les moniteurs sont de simples "consommateurs d'événements". Il n'existe aucune infrastructure qui permette à un moniteur d'influencer le déroulement de l'invocation comme par exemple l'avorter ou la rediriger vers un autre composant.

3.3 Architecture proposée dans le Java Developper's Journal

Cette dernière architecture, dont il n'existe aucune implémentation connue à ce jour, a été proposée par T. Scallan dans un article [20] paru dans la revue informatique *Java Developer's Journal*[4]. Elle est basée sur quatre composants distincts : le *Probe*, le *Profile*, le *Collector* et l'*Observer*.

Mise en œuvre

Chaque objet qui doit être monitoré possède un *Probe*. Celui-ci capture les messages échangés selon des critères qui peuvent être modifiés à tout moment et qui sont contenus dans un *Profile*.

Les données interceptées par le *Probe* sont enregistrées au niveau de l'objet pour être consultées plus tard par un autre composant appelé *Collector*. Celui-ci les retransmet à leur destination finale, l'*Observer*.

C'est ce composant qui a pour rôle de fusionner les données qui lui sont envoyées par un ensemble de *Collector*s. Les données ainsi agrégées peuvent être ensuite analysées et visualisées.

Évaluation

La différence essentielle entre cette architecture et les deux précédentes, repose sur le fait que les informations qui sont extraites des requêtes ne sont pas envoyées directement vers les moniteurs mais vers un composant intermédiaire qui les récupère. Cette manière de procéder a été choisie pour que le *Probe* ne ralentisse pas l'application.

Dans son article, l'auteur ne spécifie pas quelle est la technique employée par les *Probe* pour accéder aux messages échangés. Comme il n'existe pas d'interactivité entre les moniteurs et le processus d'interception, toutes les techniques d'inteception présentées à la fin du chapitre 2, peuvent être utilisées.

[4] http ://www.sys-con.com/java

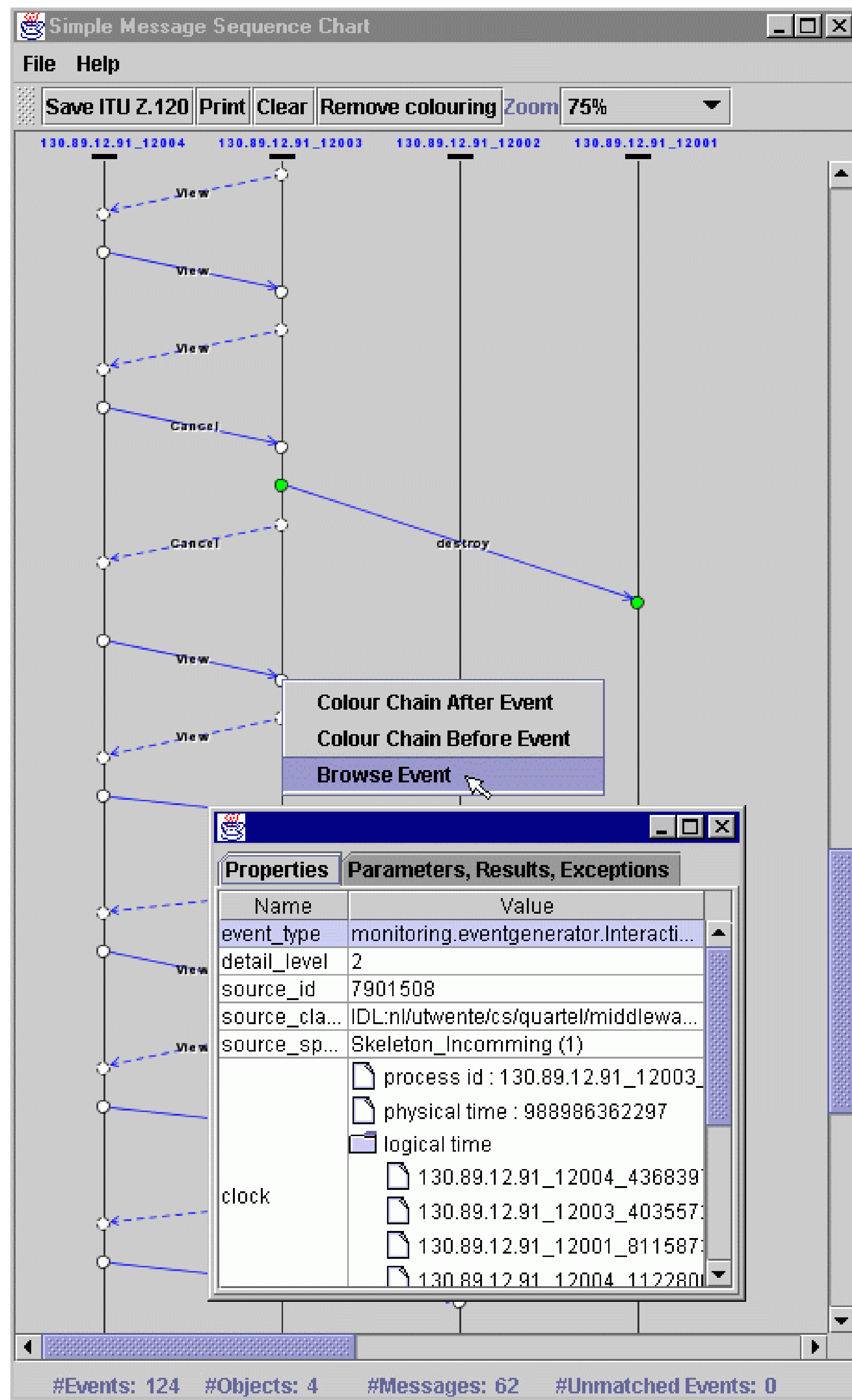

Fig. 3.2 – MODOCC : Visualisation des interactions dans un système distribué

Chapitre 4

Architecture

Nous avons présenté CORBA, défini le principe du monitoring et passé en revue d'autres projets de recherche. Nous allons maintenant aborder maintenant l'aboutissement de ce mémoire, à savoir la conception d'une architecture de monitoring de systèmes distribués.

Après avoir fixé nos objectifs et précisé nos hypothèses de travail, nous développerons l'aspect statique de notre architecture ainsi que les interactions qui existent entre les différents composants tout au long du cycle de vie de l'application.

4.1 Objectifs

Les objectifs que nous nous sommes fixés lors de la conception de notre architecture sont les suivants.

Un impact minimal sur les applications

Tout d'abord, notre architecture devait pouvoir venir se greffer sur une application qui existe déjà, avec un minimum d'effort d'adaptation.

Afin de faciliter cette intégration il nous est paru nécessaire de réduire au maximum les modifications qui doivent être apportées à l'application ainsi qu'à l'environnement dans lequel elle s'exécute.

Cette volonté de réduire l'impact sur l'application à monitorer impliquait également que notre architecture ne devait pas interférer avec d'autres services utilisés par l'application à monitorer, comme par exemple les services de transactions et de sécurité qui ont été décrits au point 1.4.4.

Une interactivité des moniteurs

Contrairement aux autres architectures que nous avons passé en revue au chapitre 3, nous avons estimé qu'il était nécessaire que les moniteurs puissent interagir sur les requêtes en temps réel.

C'est ainsi que nous avons décidé qu'un moniteur devait être au minimum, capable de faire avorter une requête s'il détecte un comportement anormal, et d'étendre cette capacité d'interaction à d'autres formes de réactions dans la mesure du possible.

Un système entièrement distribué

Nous avons souhaité réaliser une architecture de monitoring qui soit entièrement distribuée, afin de pouvoir monitorer des applications qui s'exécutent sur des machines différentes, sans devoir centraliser manuellement les informations récoltées.

Une indépendance des moniteurs par rapport au langage de programmation

Un autre objectif était de permettre d'implémenter les moniteurs indépendamment du langage de programmation utilisé pour le développement de l'application monitorée. Il en va de même pour l'implémentation de CORBA qui doit pouvoir être différente, pour des raisons de portabilité et de flexibilité.

Nous avons choisi pour arriver à cet objectif de décrire les différents composants et messages échangés par l'intermédiaire du langage IDL, ce qui nous permet de rester neutre en ce qui concerne le langage de programmation et l'implémentation de CORBA utilisée pour le déploiement des moniteurs. Il est donc possible d'utiliser le langage qui est le plus approprié pour la vérification à effectuer.

La possibilité d'intercepter des accès aux attributs

En plus de pouvoir monitorer les accès aux méthodes des serveurs, nous avons souhaité arriver à une architecture qui autorise la vérification d'accès à leurs attributs.

La réduction du temps nécessaire pour le monitoring

En ce qui concerne le monitoring, nous nous sommes fixés comme objectif de réduire son coût en terme de performance. En effet, celui-ci est proportionnel aux délais de transmission occasionnés par l'utilisation des réseaux ainsi qu'au temps nécessaire à la vérification elle-même, il dépend donc fortement de l'architecture utilisée.

4.2 Hypothèses de travail

Pour concevoir notre architecture de monitoring et en tenant compte de nos objectifs, nous avons émis des hypothèses de travail qui ont orienté sa réalisation.

Connectivité

Nous avons choisi qu'un moniteur puisse pouvoir contrôler plusieurs objets de manière à lui permettre d'assurer une fonction de synchronisation.

En même temps, nous avons souhaité qu'un objet puisse être monitoré par plusieurs moniteurs afin que ceux-ci puissent être spécialisés par rapport à certaines fonctions de monitoring de l'objet.

Accessibilité des acteurs

Le client n'étant pas toujours sous notre contrôle, nous avons décidé de ne pas travailler à son niveau. Ce choix implique que les fonctionnalités de monitoring soient introduites uniquement au niveau du serveur.

Distribution géographique des composants

Une autre hypothèse concerne la distribution des différents composants, nous l'avons considérée de la manière suivante :

- Les différents moniteurs qui assurent le rôle de surveillance d'une ou de plusieurs objets sont considérés comme géographiquement proches.
- Les serveurs sont considérés comme étant géographiquement éloignés des moniteurs

L'impact de la distance qui sépare les différents composants se fait surtout ressentir dans les délais de latence lors des communications ainsi qu'au niveau de la vitesse de transfert des informations.

Cette hypothèse nous permettra d'améliorer les performances de notre architecture en introduisant un nouveau composant.

4.3 Description générale de l'architecture

Avant de développer les différents composants et les mécanismes de fonctionnement de notre architecture, il nous semble indispensable de présenter une description schématique et succincte de son organisation.

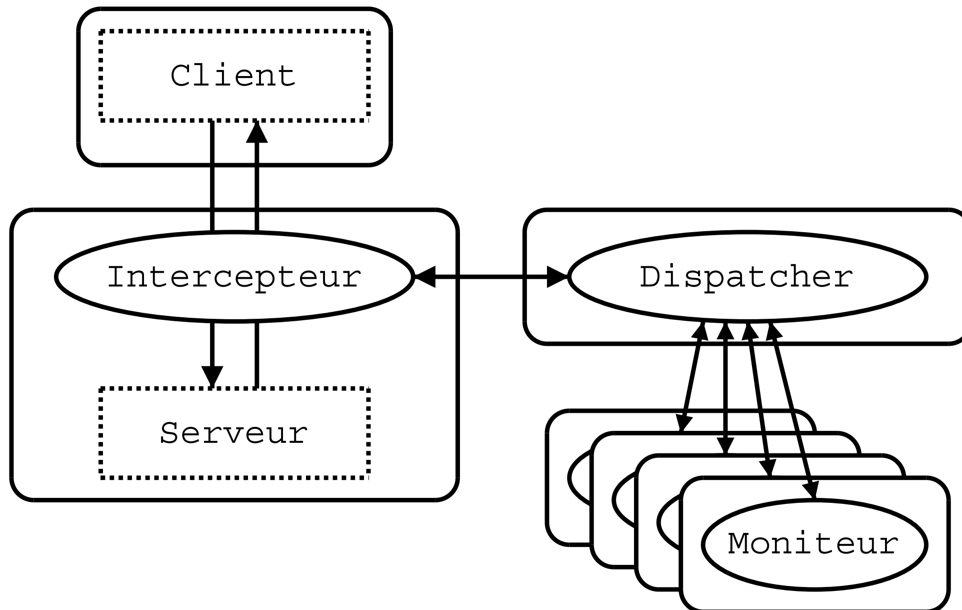

Fig. 4.1 – Architecture

Comme le montre la figure 4.1, l'architecture développée dans ce mémoire repose sur trois composants principaux :

- les intercepteurs qui sont situés au niveau des serveurs et qui ont pour fonction d'intercepter des requêtes qui leurs sont adressées ainsi que les réponses qu'ils renvoient.
- un ensemble de moniteurs qui sont chargés d'analyser ces informations afin d'en déterminer la validité et d'éventuellement agir en conséquence.
- un dispatcheur qui a pour mission essentielle d'assurer la communication entre les intercepteurs et les moniteurs.

4.4 Choix du procédé d'interception

Étant donné les objectifs que nous nous sommes fixés et les différentes techniques exposées au point 2.3, nous avons choisi d'utiliser les intercepteurs CORBA pour intercepter les messages reçus et émis par un serveur.

Comme on peut le voir dans le tableau 4.2 qui reprend les fonctionnalités et inconvénients des différentes techniques d'interception, ce sont les intercepteurs CORBA qui ont le meilleur rapport avantages/inconvénients. Nous évoquons succinctement les raisons qui nous ont amené à ne pas envisager une autre méthode :

- Le *sniffing* ne permet pas d'interaction de la part des moniteurs et peut nécessiter une modification de l'architecture du réseau.

	Sniffing	OS	Stub/Skeleton	Proxy	Intercepteurs
Interactif	Non	Oui	Oui	Oui	Oui
Acces aux attributs	Oui	Oui	Oui	Non	Oui
Transparent niveau client					
Modification	Non	Non	Non	Oui	Non
Recompilation	Non	Non	Oui	Oui	Oui
Transparent niveau serveur					
Modification	Non	Non	Non	Oui	Non
Recompilation	Non	Non	Oui	Oui	Oui
Portable	Oui	Non	Non	Oui	Oui
Requetes dynamiques	Oui	Oui	Non	Oui	Oui

Tab. 4.2 – Caractéristiques des différentes techniques d'interception

- L'interception au niveau du système d'exploitation est trop dépendante de celui-ci, ce qui le rend inutilisable dans le cadre d'un système hétérogène.
- Les stubs et skeletons modifiés présentent l'inconvénient majeur d'être liés à une implémentation de CORBA, ce qui ne permet pas leur utilisation dans le cadre d'une application existante qui utilise des ORBs différents.
- L'introduction de *proxy*s nous aurait limité à l'interception des requêtes, de plus ils nécessitent d'être adaptés à l'objet qui est monitoré afin de pouvoir modifier les références renvoyées.

Minimalisation de la complexité de l'intercepteur

Un choix que nous avons posé lors du design de notre architecture était celui de minimaliser la complexité de l'intercepteur pour les raisons suivantes :

- L'intercepteur est le seul composant qui dépend du langage employé pour l'implémentation du serveur. Réduire sa complexité permet de ne pas devoir réécrire une grande quantité de code lorsqu'il doit être associé à une application qui est programmée dans un langage de programmation différent.
- Limiter sa complexité permet également de rester indépendant de l'ORB utilisé en n'ayant recours qu'aux fonctionnalités de base qui sont définies par l'OMG.
- Il est par ailleurs indispensable que le processus d'interception ait un impact minimal sur les performances des autres applications exécutées sur la même machine.

4.5 Description des composants

Nous allons maintenant aborder la description des différents composants de notre architecture, ainsi que leur fonctionnement interne. Nous allons approfondir maintenant chaque composant de façon indépendante en considérant les autres comme des "boîtes noires". Les détails qui concernent les interactions seront explicités à la fin de ce chapitre.

4.5.1 L'intercepteur au niveau du serveur

Comme nous l'avons énoncé dans nos hypothèses, c'est uniquement au niveau de l'intercepteur du serveur que se fait le processus d'interception des requêtes et des résultats.

Des cinq points d'interception de cet intercepteur, seul le `receive_request_service_contexts` n'est pas utilisé car il n'est pas fait appel aux fonctionnalités offertes par les contextes de service. Il n'y a donc pas de transfert d'informations entre les clients et les serveurs.

Nous allons développer les différentes opérations qui sont effectuées par l'intercepteur lors de la réception d'une requête, l'envoi d'un résultat ou l'émission d'une exception.

Interception d'une requête

Toute invocation de requête portant sur un objet provoque en premier lieu l'activation du `receive_request` de l'intercepteur.

A ce moment, il effectue une recherche dans son répertoire d'interfaces afin de déterminer si la requête doit être monitorée. Si le résultat de cette consultation est négatif, l'interception se termine et l'invocation suit cours.

Si le résultat de la recherche est positif, les informations concernant la requête sont dupliquées et stockées dans une structure identique à celle décrite en IDL dans le listing 4.1. Cette structure est ensuite envoyée de façon asynchrone au *Dispatcher* qui la retransmet à tous les moniteurs intéressés.

L'intercepteur reprend son exécution lorsque tous les moniteurs ont terminé d'analyser les différentes informations qui leur ont été transmises par l'intermédiaire de cette structure.

Si une exception a été soulevée pendant l'analyse, que ce soit par CORBA ou par un moniteur, une exception système est renvoyée à l'objet qui est à l'origine de la requête. L'invocation de la requête est donc avortée et le client est averti de l'erreur.

Si aucune exception n'a été soulevée, l'intercepteur sauvegarde dans un conteneur associatif la structure qui lui a été transmise. La clé à laquelle sont associées ces

Listing 4.1 – Structure du RequestInformations

```
10 struct Time
11 {
12   long sec;
13   long usec;
14 };
15
16 struct RequestInformations
17 {
18   unsigned long interceptorId;
19   unsigned long requestId;
20   CORBA::RepositoryId repositoryId;
21   string operation;
22   Object target;
23
24   boolean expectAnswer;
25   Dynamic::ParameterList parameters;
26   Time timeStamp;
27 };
```

données est l'identificateur de requête qui est attribué par CORBA. Comme cet identificateur est identique pour une requête et sa réponse, il sera possible de récupérer ces informations plus tard, lors de l'interception de la réponse du serveur.

Dès que cette procédure de sauvegarde est terminée, la requête est effectuée par le serveur comme si elle n'avait jamais été interceptée.

Interception de la réponse

Lorsque l'opération exécutée par le serveur se termine sans soulever d'exception, l'intercepteur est réactivé au point `send_reply`, avant que le résultat ne soit transmis au client.

La nécessité de monitorer la réponse est déterminée au moyen du conteneur associatif dans lequel les informations ont été stockées lors de la réception de le requête : s'il existe une entrée dans le conteneur ayant pour clé l'identificateur de requête c'est que la réponse doit être monitorée.

Dans les informations mises à la disposition d'un intercepteur se trouve un identificateur de requête. Celui-ci étant identique pour une requête et sa réponse, il est possible de déterminer si l'invocation a été monitorée et de récupérer les informations qui ont été envoyées au moniteur.

Grâce à ce transfert de données entre le `receive_request` et `send_reply` on peut obtenir à ce point d'interception le nom de l'interface qui normalement n'est pas

Listing 4.2 – Structure du ReplyInformations

```
29 struct ReplyInformations
30 {
31   unsigned long interceptorId;
32   unsigned long requestId;
33   CORBA::RepositoryId repositoryId;
34   string operation;
35   Object target;
36
37   any result;
38   Time elapsedTime;
39 };
```

disponible à cet endroit[1].

Avec les informations qui concernent la requête et celles disponibles à ce point d'interception, une structure de données qui décrit la réponse est construite et envoyée aux moniteurs par l'intermédiaire du dispatcheur. Cette structure est décrite dans le listing 4.2.

Tout comme lors de la vérification de la requête, un moniteur peut soulever une exception. Celle-ci a pour conséquence de ne pas renvoyer les résultats au client et de générer une exception chez celui-ci.

Au contraire, quand le résultat est validé par l'ensemble des moniteurs, les résultats sont alors transmis au client.

Exceptions survenues au niveau du serveur

Un serveur peut également soulever une exception pendant l'exécution de l'opération demandée. Il en résulte, selon le type de l'exception, l'activation du point `send_exception` ou `send_other` de l'intercepteur. Celui-ci va alors transmettre cet événement au dispatcheur, qui le retransmettra aux moniteurs.

De manière générale, nous savons que lorsqu'une exception a été soulevée par le serveur il n'y aura pas de réponse émanant de celui-ci. L'intercepteur efface dès lors la structure qu'il avait sauvegardée lors du `receive_request` afin d'éviter une fuite de mémoire.

4.5.2 L'accès à la configuration des intercepteurs

Comme nous l'avons évoqué au point 1.4, les intercepteurs sont des objets CORBA locaux. Ils sont dès lors inaccessibles depuis un objet distant. Cette limitation en-

[1] voir tableau 1.2

Listing 4.3 – Interface IDL de l'accesseur

```
3  typedef sequence<CORBA::RepositoryId> RepositoryIdSeq;
4
5  interface Accessor
6  {
7    void addInterfaces(in RepositoryIdSeq ris);
8    void removeInterfaces(in RepositoryIdSeq ris);
9
10   void print(in string s);
11 };
```

Listing 4.4 – Interface IDL du RequestMonitor

```
47 interface RequestMonitor
48 {
49   ReplyMonitor request(in RequestInformations ri)
50     raises (MonitorException);
51 };
```

traîne l'impossibilité de changer de manière dynamique le comportement d'un intercepteur au cours de l'exécution de l'application monitorée.

La solution retenue pour contourner cette impossibilité est de leur associer un autre objet non local qui a accès à l'intercepteur. C'est ce nouvel objet, dénommé *accesseur*, qui permet un accès à distance à la configuration de l'intercepteur. Pour ce faire il possède deux méthodes : `addInterface` et `removeInterface`. Celles-ci permettent respectivement d'ajouter et de retirer de l'intercepteur des interfaces à monitorer.

L'interface de l'accesseur est décrite dans le listing 4.3.

4.5.3 Les moniteurs

Les moniteurs sont composés de deux entités : les moniteurs de requêtes et les moniteurs de réponses.

Les moniteurs de requêtes

Les moniteurs de requêtes, ou *RequestMonitor*s, sont des objets distants qui vont analyser les informations[2] qui leur sont envoyées par les intercepteurs lors de la réception d'une requête. La description IDL de ces moniteurs est donnée au listing 4.4.

[2]décrites dans le listing 4.1

Listing 4.5 – Interface IDL du ReplyMonitor

```
41 interface ReplyMonitor
42 {
43   void reply(in ReplyInformations ri)
44     raises (MonitorException);
45 };
```

Si le moniteur décide sur base de ces informations reçues que la requête est valide, il peut créer un *ReplyMonitor* pour indiquer qu'il désire monitorer la réponse de celle-ci. La création de cet objet lui permet également de sauvegarder le contexte de l'invocation pour une consultation ultérieure. Si le moniteur renvoie une référence nulle cela signifie qu'il a validé la requête mais qu'il n'est pas intéressé par la réponse.

Si le moniteur considère que la requête est invalide, il a la possibilité d'empêcher celle-ci de parvenir au serveur en soulevant une exception de type *MonitorException*. Celle-ci est retransmise aux intercepteurs par le biais du dispatcheur, accompagnée d'un message qui décrit la raison qui a mené à l'interruption de la requête.

Notons qu'un moniteur de requête ne recevra jamais d'informations concernant des interfaces pour lesquelles il n'a pas annoncé l'intérêt auprès des intercepteurs. Cette annonce se fait lors de l'enregistrement du moniteur auprès du *dispatcher*. Nous verrons plus loin comment cette opération est réalisée.

Le moniteur de réponse

Le *ReplyMonitor*, ou moniteur de réponse, est un objet qui est associé à chaque requête dont la réponse doit être monitorée. Lorsqu'une réponse émise par un serveur est interceptée par le point `send_reply` de l'intercepteur, il a pour fonction d'en vérifier la validité au moyen de sa méthode `reply`.

Comme on peut le voir dans le listing 4.5, il dispose pour cette vérification des informations qui correspondent à la structure de données transmise par l'intercepteur lors du `send_reply`. A celles-ci viennent également se rajouter toutes les données sauvegardées par le *RequestMonitor* lors de leur création. Il s'agit généralement d'informations qui concernent la requête.

Tout comme le moniteur qui vérifie l'exactitude de la requête, le moniteur de réponse peut soit provoquer une exception si le résultat de la vérification est incorrect, soit rendre la main au programme appelant.

Nous précisons également que la durée de vie de ces *ReplyMonitor*s est limitée au monitoring d'une seule requête : ils sont détruits dès que leur fonction de vérification s'est achevée.

Listing 4.6 – Interface IDL du dispatcheur

```
6  interface Dispatcher
7  {
8    void request(in RequestInformations ri) raises (MonitorException);
9    void reply(in ReplyInformations ri) raises (MonitorException);
10
11   unsigned long registerInterceptor(in Accessor ic);
12   void registerMonitor(in RequestMonitor mm, in RepositoryIdSeq ris);
13 };
```

4.5.4 Le dispatcheur

Le dispatcheur est le composant qui assure la transmission des informations entre un intercepteur et un ensemble de moniteurs.

Notre choix d'introduire cet intermédiaire est établi sur l'hypothèse qui concerne la distribution géographique des composants, telle que nous l'avons énoncée au point 4.2. En effet, il permet de ne transmettre qu'une seule fois des informations au travers de la section la plus lente et de les redistribuer ensuite aux différents moniteurs, ce qui augmente les performances du monitoring

Comme nous pouvons le voir dans le listing 4.6, le dispatcheur dispose de quatre méthodes. Deux d'entre elles, `registerMonitor` et `registerInterceptor`, servent à l'enregistrement de nouveaux moniteurs et intercepteurs auprès du dispatcheur. L'utilité de ces fonctions sera explicitée plus loin lorsque nous parlerons des procédures d'enregistrement au point 4.6.

Le deux autres méthodes, à savoir `request` et `reply`, sont utilisées pour le monitoring des requêtes lorsqu'un intercepteur désire transmettre des informations aux moniteurs.

Monitoring d'une requête

Dans le cas d'une requête, il s'agit bien évidemment de la méthode `request` qui est invoquée sur le dispatcheur. Dans les données qui lui sont transmises figure le nom de l'interface, ce qui lui permet de déterminer à quels moniteurs il doit retransmettre les données reçues. Pour effectuer sa sélection, il dispose d'une base de données telle celle qui est représentée à la figure 4.2.

Afin de réduire le délai occasionné par l'exécution des moniteurs, nous avons décidé que celle-ci devait être réalisée en parallèle. Nous avons rejeté une première approche qui utilisait les invocations asynchrones en raison de la non-fiabilité de ce type d'invocation mais aussi parce qu'il était impossible d'obtenir un résultat de la part des moniteurs.

Ces problèmes auraient pu être résolus par l'utilisation de *callback*s qui auraient

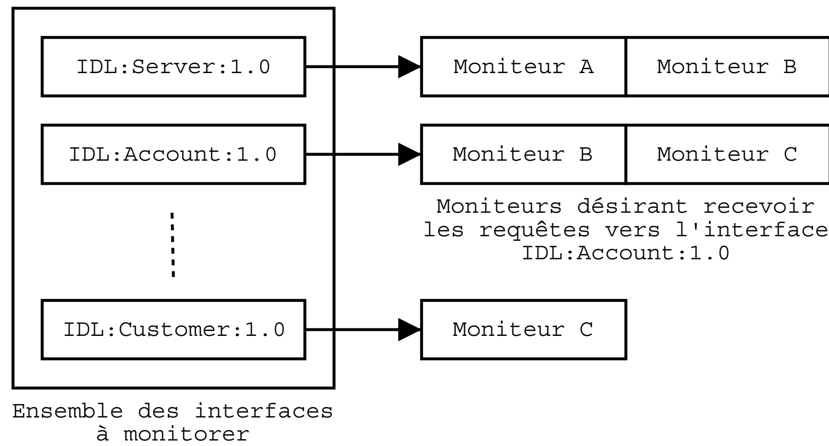

FIG. 4.2 – Base de données des interfaces et moniteurs

permis d'une part aux moniteurs d'avertir le dispatcher de leur fin d'exécution et d'autre part à ce dernier de s'assurer de la bonne réception des requêtes par les moniteurs. Cette solution ne nous a néanmoins pas convaincu, principalement en raison de la complexité de sa mise en place.

La solution que nous avons finalement retenue est l'utilisation du *multi-threading* qui offre la possibilité d'exécuter plusieurs méthodes de manière concurrente. Dans le dispatcheur, chaque invocation de méthode sur un moniteur se fait dans un *thread* indépendant, ce qui nous permet d'arriver à notre objectif sans perdre de fonctionnalités et sans augmenter considérablement sa complexité.

Comme nous l'avons vu lors de la présentation des *Monitor*s, ceux-ci renvoient à la fin de leur exécution une référence vers un *ReplyMonitor* s'ils désirent être informés du résultat de la requête qui est retourné par le serveur. Ces références sont sauvegardées dans une autre base de données, qui a comme identifiant un couple composé de l'identificateur unique de l'intercepteur ainsi que de l'identificateur de la requête auprès de cet intercepteur.

Les différentes interactions entre les composants qui ont lieu lors du monitoring d'une requête validée sont représentées à la figure 4.3 sous la forme d'un diagramme de séquencement.

Il nous reste encore à détailler le comportement du dispatcheur si celui-ci reçoit une ou plusieurs exceptions soulevées par des *Monitor*s. Dans ce cas, les messages qui sont contenus dans les *MonitorException*s sont rassemblés en une seule chaîne de caractères, qui est ensuite renvoyée à l'intérieur d'un nouveau *MonitorException* à l'intercepteur. Les *ReplyMonitor*s qui ont été créés suite à la validation de la requête par d'autres moniteurs sont détruits.

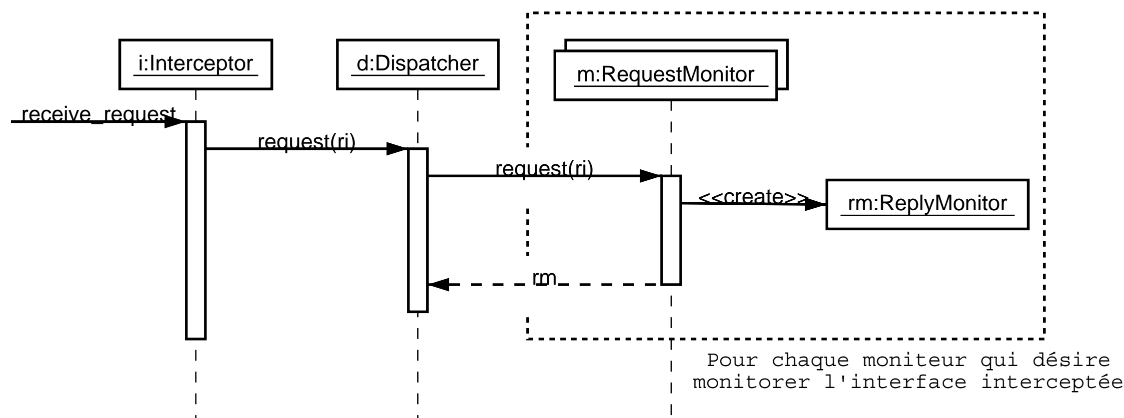

Fig. 4.3 – Monitoring d'une requête

Monitoring de la réponse

Lorsqu'un intercepteur intercepte une réponse provenant d'un serveur qui doit être monitoré, celui-ci la retransmet au dispatcher au moyen de l'invocation de sa méthode `reply`. Le dispatcheur utilise alors les informations qui lui sont envoyées sous la forme de paramètres afin de retrouver les *ReplyMonitor*s qui ont été sauvegardés lors du `request`.

Dès que ces *ReplyMonitor*s ont été récupérés, le dispatcheur peut leur transmettre la réponse qu'il a reçu. Tout comme lors du `request`, ces opérations sont effectuées dans des *thread*s différents afin qu'elles puissent être exécutées en parallèle. Lorsque chaque *ReplyMonitor* a terminé son exécution, ceux-ci sont détruits.

4.6 Enregistrement des intercepteurs et des moniteurs

Nous avons vu dans la section précédente que le dispatcheur doit disposer de renseignements concernant les intercepteurs et les moniteurs :

- Ceux qui concernent les intercepteurs sont nécessaires pour pouvoir les actualiser, ils sont constitués de la référence vers leurs *accesseurs*.
- Ceux qui concernent les moniteurs sont indispensables pour pouvoir leur adresser les événements envoyés par les intercepteurs, il s'agit de leurs références ainsi que la liste des interfaces qu'ils supportent.

Pour obtenir ces différentes informations, les intercepteurs ainsi que les moniteurs doivent s'incrire auprès du dispatcheur. Dans cette section, nous allons aborder les modalités de cette inscription.

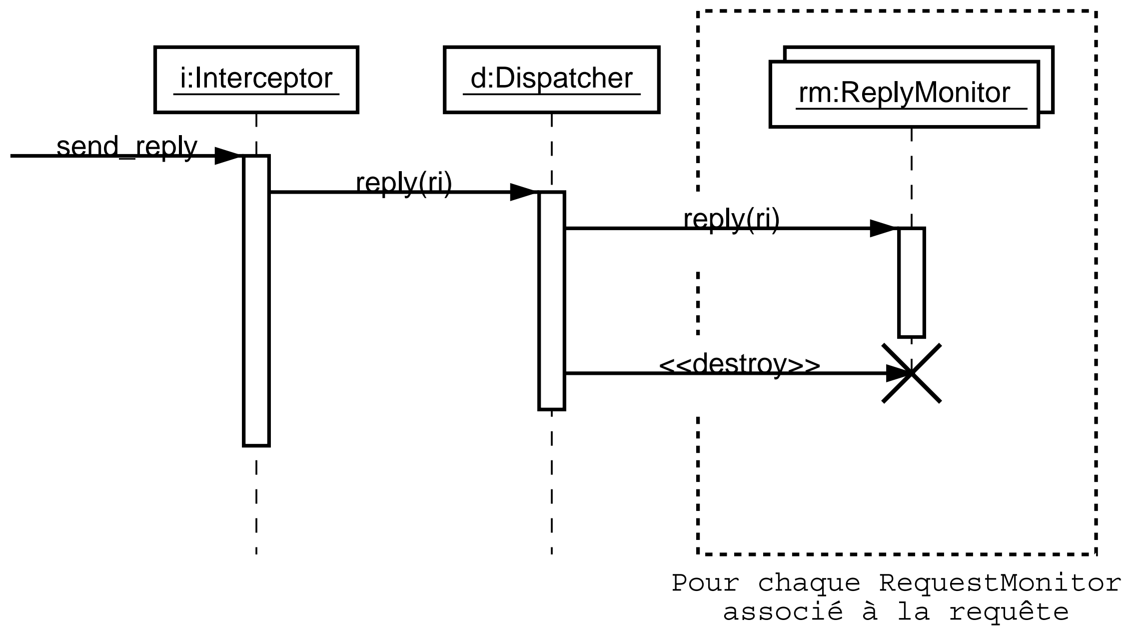

FIG. 4.4 – Monitoring de la réponse

4.6.1 L'enregistrement d'un intercepteur

L'inscription d'un intercepteur auprès d'un dispatcheur présente une triple fonction :

- d'une part, elle permet au moniteur de prendre connaissance de l'*accesseur* associé à cet intercepteur
- d'autre part de transmettre à l'intercepteur la liste des interfaces à monitorer
- ensuite d'attribuer à chaque intercepteur un numéro qui l'identifie auprès du dispatcheur et qui est utilisé lors de la sauvegarde des *ReplyMonitor*s.

Lors de son initialisation, l'intercepteur crée en premier lieu un *accesseur*. La référence de celui-ci est ensuite transmise au dispatcher au moyen de l'invocation de sa méthode `registerInterceptor`.

La première action qui est effectuée par le dispatcher est de transmettre à l'intercepteur, par l'intermédiaire de son *accesseur*, la liste des interfaces qu'il doit monitorer. Cette liste est établie à partir de l'ensemble des interfaces qui ont été enregistrées dans la base de données du dispatcher.

Après avoir sauvegardé la référence de l'accesseur qui lui a été transmise par l'intercepteur, le dispatcheur génère un numéro d'identification unique (*InterceptorId*) qui lui est renvoyé.

Le diagramme de séquencement représentant ces différentes opérations est présenté à la figure 4.5.

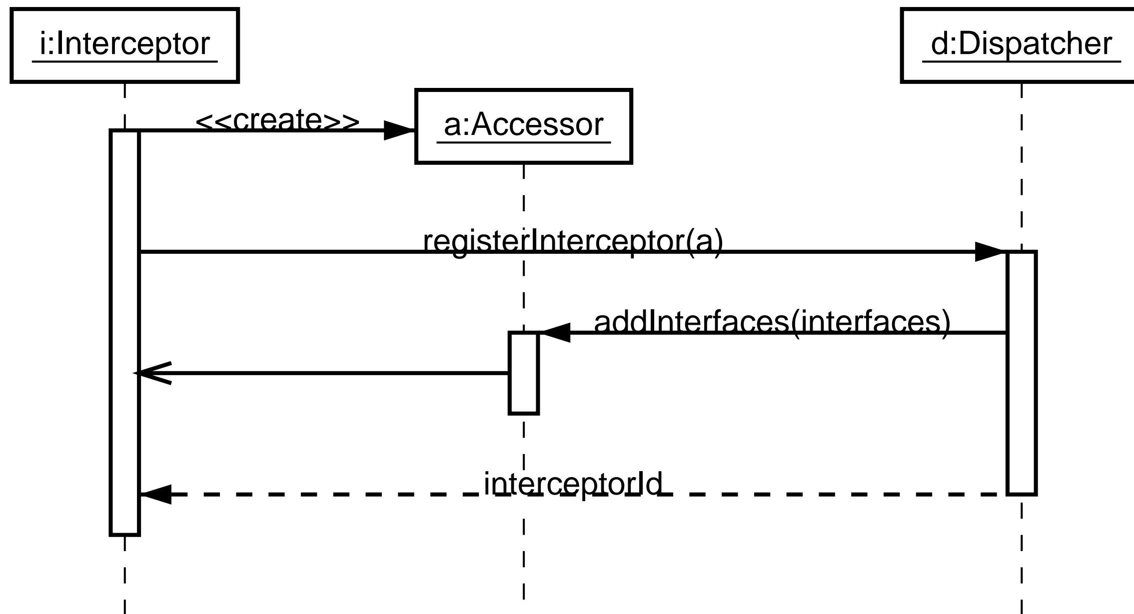

FIG. 4.5 – Initialisation d'un intercepteur

4.6.2 L'enregistrement d'un moniteur

Tout comme les intercepteurs, les moniteurs doivent s'inscrire auprès d'un dispatcheur afin d'annoncer quelles sont les interfaces qu'ils souhaitent voir interceptées.

Les informations qu'ils transmettent pour cette opération sont leurs références ainsi qu'une liste de noms d'interfaces. Ces deux informations sont sauvegardées dans la base de données du dispatcheur utilisée lors de la réception d'une requête.

Le dispatcheur va ensuite se connecter sur tous les accesseurs des intercepteurs qui se sont enregistrés auprès de lui afin de leur annoncer les nouvelles interfaces à intercepter.

Les différentes interactions qui sont décrites ci-dessus peuvent être présentées sous la forme du diagramme de séquencement de la figure 4.6.

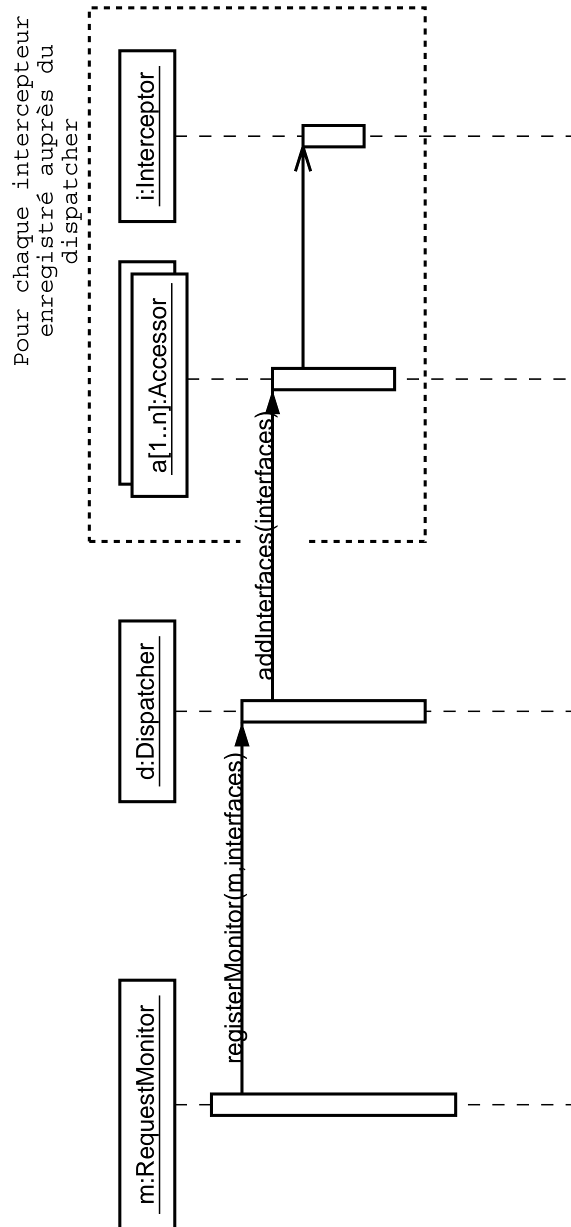

Fig. 4.6 – Initialisation d'un moniteur

Chapitre 5

Prototype

Il nous a paru indispensable de valider la faisabilité de notre approche et de mesurer son impact sur les performances de l'application monitorée. Nous avons donc réalisé un prototype qui se présente sous la forme d'un intercepteur et d'un dispatcheur.

5.1 Choix de l'implémentation CORBA

La première étape dans l'implémentation étant le choix d'une implémentation de CORBA, nous avons déterminé deux exigences. D'une part, le langage de cette implémentation doit être orienté objet et doit produire des exécutables efficaces. D'autre part, elle doit être compatible avec les spécifications des intercepteurs CORBA.

En ce qui concerne notre première exigence, notre sélection s'est portée sur le langage C++ qui répond mieux à nos critères comparé à d'autres langages similaires, tel que le Java. Nous avions tout d'abord retenu une implémentation de CORBA réalisée dans ce langage et qui est connue sous le nom de *Mico*[1]. Malheureusement, après avoir lu les exemples relatifs aux intercepteurs qui étaient fournis avec celle-ci il nous est apparu qu'elle n'était pas conforme aux spécifications de l'OMG qui concernent les *Portable Interceptors*. Ceci la rendait inutilisable dans le cadre du développement de notre projet.

Un second choix s'est alors porté sur *The ACE ORB* (TAO)[2]. Cependant, la dernière version disponible lors des études des différents ORBs n'était également pas compatible avec l'ensemble des spécifications formant CORBA 2.3, notamment en ce qui concerne les intercepteurs.

En définitive, nous avons sélectionné *ORBacus*[3] de la société IONA. Les raisons qui nous ont amené à effectuer ce choix final sont les suivantes :

[1] http ://www.mico.org
[2] http ://www.cs.wustl.edu/ schmidt/TAO.html
[3] http ://www.iona.com/products/orbacus_home.htm

– il est disponible pour C++ et accessoirement pour Java
– il est conforme à la version 2.4 des spécifications de CORBA, y compris les *Portable Interceptors*
– il supporte le *multi-threading*
– il met à la disposition des développeurs un *Naming Service* prêt à l'emploi
– il a de hautes performances lorsque les objets sont situés sur une même machine

De nouvelles orientations dans la politique commerciale d'IONA ont eu pour conséquence le retrait du compilateur IDL des sources d'*ORBacus* et l'instauration pour celui-ci d'un système de licence. Malgré la possibilité d'octroi de licences "académiques", des problèmes de compatibilité binaire nous ont contraint à utiliser une version antérieure[4].

5.2 Implémentation

L'implémentation des intercepteurs ainsi que celle du dispatcheur est reproduite en annexe en raison du nombre élevé de fichiers et de leur taille. Voici une liste qui décrit les différents modules et qui permet de les situer dans les annexes.

- l'`ORBInitializer` (p81-82) est l'objet qui permet d'initialiser l'ORB. C'est dans celui-ci que sont enregistrés les intercepteurs.
- le `ServerRequestInterceptor` (p83-88) est l'implémentation de l'intercepteur conforme aux spécifications 2.3 de CORBA, qui envoie les différents évènements au *Dispatcher*. Il contient également une procédure qui permet de s'inscrire auprès de celui-ci.
- l'`Accessor` (p89-90) offre aux objets distants la possibilité d'accéder à un intercepteur. Son utilisation est détaillée dans le chapitre 4.
- la classe `Dispatcher` (p91-98) implémente le dispatcheur tel que nous l'avons décrit dans la section 4.5.4 de la description de notre architecture.

5.3 Mode d'emploi

La mise en place du monitoring pour un serveur se fait en deux étapes. En premier lieu, des modifications doivent être apportées à l'initialisation de celui-ci afin de mettre en place le processus d'interception des requêtes. Cette opération effectuée, il ne reste qu'à implémenter les moniteurs selon les objectifs que l'on souhaite atteindre.

Pour illustrer ces opérations, nous avons implémenté les interfaces *PrintServer* et *PrintSession* qui ont été utilisées comme exemples de déclarations IDL dans le pre-

[4] version 4.1.0

Listing 5.1 – Initialisation de l'intercepteur

```
7  #include <PrintServer_impl.h>
8  #include <ORBInitializer_impl.h>
9
10 int main(int argc, char* argv[])
11 {
12   // Initialise l'intercepteur et l'ORB
13   CORBA::ORB_var orb = ORBInitializer_impl::ORB_init(argc,argv);
```

mier chapitre. Les fonctionnalités que nous avons intégrées dans ce serveur sont minimales et se résument aux éléments suivants :

- l'opération `connect` vérifie le mot de passe dans la base de données contenant les utilisateurs. S'il est correct elle crée un objet de type `PrintSession` et le renvoie à l'utilisateur. Dans le cas contraire, une référence nulle est retournée.
- la méthode `addUser` permet de rajouter des utilisateurs dans la base de données utilisée par le `connect`. La réponse est un booléen : *false* si l'utilisateur était déjà inscrit, *true* dans le cas contraire.
- lorsqu'un utilisateur invoque un `print` sur la session obtenue lors de sa connexion, le message est affiché à l'écran au lieu d'être envoyé à l'imprimante.
- un `reset` provoque également l'affichage d'un message.
- finalement, lors d'un `disconnect`, l'objet `PrintSession` sur lequel l'opération a été invoquée est détruit.

Le code source de ces deux objets est fourni en annexe, de la page 101 à 103.

5.3.1 Adaptation du serveur

Pour pouvoir monitorer l'activité de ce serveur, deux opérations sont nécessaires.

Tout d'abord, les intercepteurs CORBA doivent être créés et enregistrés auprès de l'ORB. Suivant les spécifications de l'OMG, cette opération doit impérativement être effectuée avant l'initialisation du bus CORBA. Ces intercepteurs doivent ensuite être enregistrés auprès d'un *Dispatcheur*, dont la localisation est assurée au moyen du *Naming Service*[5].

Afin de minimiser l'importance des modifications à apporter au serveur, nous avons remplacé la fonction `CORBA::ORB_init` par une autre fonction qui initialise notre intercepteur et ensuite l'ORB. Celle-ci porte le nom de `ORBInitializer_impl::ORB_init`. Elle a une signature identique à la fonction initiale ce qui les rend interchangeables.

[5] voir 1.3

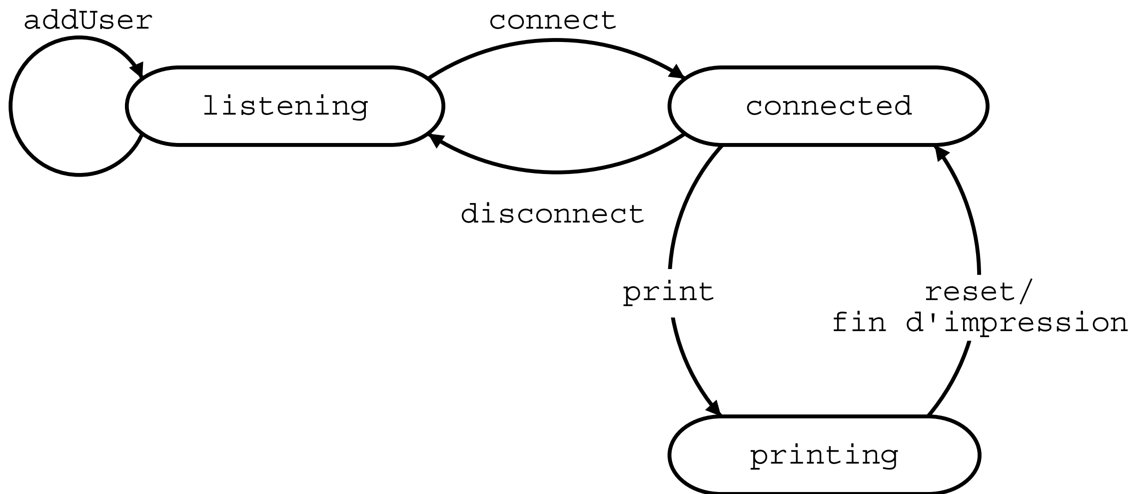

Fig. 5.1 – Machine à états du serveur d'impression

Le plus souvent, les modifications à apporter au serveur se résument donc à un changement de nom de fonction ainsi qu'à l'inclusion du fichier `ORBInitializer_impl.h` qui contient la déclaration de notre nouvelle méthode.

Le fragment de code du listing 5.1 qui contient l'initialisation de notre serveur d'impression est extrait du fichier `PrintServer_main.cpp`[6]. On peut remarquer l'inclusion du fichier `ORBInitializer_impl.h` à la ligne 8, ainsi que l'initialisation de l'intercepteur à la ligne 13.

5.3.2 Les moniteurs

Les moniteurs sont les composants qui vont traiter les différents événements qui leur sont transmis. Lors de la présentation de notre architecture nous avons expliqué qu'ils sont constitués de deux entités : d'une part les *RequestMonitor*s qui analysent les requêtes et d'autre part les *ReplyMonitor*s qui vont analyser les réponses correspondantes

Afin d'étendre les fonctionnalités de notre serveur d'impression sans devoir y apporter de modifications, nous avons réalisé un petit moniteur qui rend possible :

- La gestion d'un *timeout* pour les sessions ce qui permet d'empêcher toute opération sur celles-ci après un délai d'inactivité qui est configurable dans le moniteur.
- La vérification de la taille du mot de passe lors de la création d'un nouvel utilisateur. Si celui-ci est trop court, l'opération est avortée.
- La vérification du respect de la machine à états qui est représentée à la figure 5.1. Il s'agit donc de vérifier, pour chaque fonction qui représente une

[6]fichier en annexe, page 102

transition, si elle a bien eu lieu dans la session d'impression telle qu'elle est spécifiée dans ce diagramme.

C'est ce moniteur nouvellement créé qui servira de support pour les explications relatives à l'implémentation d'un moniteur qui utilise notre architecture. L'intégralité du fichier peut être consulté à la page 104.

Le *RequestMonitor*

Le monitoring des informations relatives à une requête est effectué par la fonction `request` qui est définie dans l'interface `RequestMonitor`[7]. L'argument de cette fonction est une structure qui contient l'ensemble des informations concernant la requête. Ce sont celles-ci qui sont utilisées par le moniteur afin de déterminer l'interface et l'opération qui ont provoqué son invocation.

```
ReplyMonitor_ptr RequestMonitor_impl::request(const RequestInformations& ri)
  throw(MonitorException,
        CORBA::SystemException)
{
  if INTERFACE("IDL:PrintServer:1.0")
    {
      return checkRequest_PrintServer(ri);
    }
```

L'extrait ci-dessus reprend les premières lignes de notre fonction réalisant le monitoring d'une requête. On peut y remarquer que lorsque l'événement porte sur une interface `PrintServer`, la fonction `checkRequest_PrintServer` qui est décrite ci-dessous est exécutée et que son résultat est retourné au dispatcheur.

```
ReplyMonitor_ptr checkRequest_PrintServer(const RequestInformations& ri)
  throw(MonitorException)
{
  // Extraction des paramètres de la requête
  const char* user;
  assert(ri.parameters[0].argument >>= user);
  const char* password;
  assert(ri.parameters[1].argument >>= password);

  // Teste si l'opération est addUser
  if (OPERATION("addUser"))
    {
      // Vérifie la taille du mot de passe, soulève une exception
      // si celui-ci est trop court. Cette erreur est également
      // loggée.
```

[7] voir listing 4.4

```
      if ( strlen(password) < MIN_PASSWORD_LENGTH)
        {
          LOG_SERVER << "Aborting user creation, password for user '"
                     << user << "' is too short "
                     << "(size=" << strlen(password) << ")" << endl;
          throw MonitorException("Password too short");
        }
    }

  // Nous désirons inscrire un moniteur qui va inspecter les réponses
  // pour les deux opérations du PrintServer
  ReplyMonitor_impl* rmi = new ReplyMonitor_impl(user);
  return rmi->_this();
}
```

La première opération qui est effectuée par cette fonction `checkRequest_PrintServer` est l'extraction des paramètres de l'invocation qui sont communs aux deux méthodes supportées par l'interface `PrintServer`.

Elle détermine ensuite s'il s'agit de l'opération `addUser` qui a été invoquée sur le serveur et dans ce cas, elle vérifie la taille du mot de passe. S'il a une taille inférieure à une valeur fixe, appelée ici `MIN_PASSWORD_LENGTH`, une exception est soulevée.

Si le mot de passe a une taille supérieure ou que l'opération est un `connect`, un objet `ReplyMonitor` est créé et le nom de l'utilisateur est sauvegardé dans ce dernier car les arguments de la requête ne seront plus disponibles lors de la réception de la réponse.

Cette référence est donc retournée à la fonction `request`, qui la retransmet ensuite au dispatcheur afin que la réponse puisse être monitorée.

Le *ReplyMonitor*

Ce *ReplyMonitor* contient une seule fonction appelée *reply*, dont le rôle est de valider la réponse du serveur avant que celle-ci ne soit renvoyée au client. Les informations dont elle dispose sont celles qui lui ont été transmises en argument. A celles-ci viennent s'ajouter celles qui ont été sauvegardées lors de la création du *ReplyMonitor* par le *RequestMonitor*.

L'extrait ci-dessous reprend les opérations qui sont effectuées par notre moniteur pour les réponses émanant des méthodes `addUser` et `connect` du *PrintServeur*.

```
  if OPERATION("connect")
    {
      PrintSession_ptr session;
      assert(ri.result >>= session);
```

```
    if (! CORBA::is_nil(session))
      {
        LOG_SERVER << "Connection for user '" << save_user << "'"
                   << ", referring as PrintSession[" << id_counter
                   << "]" << endl;
        SessionInfo *si = new SessionInfo;
        si->user = strdup(save_user);
        si->state = IDLE;
        si->id = id_counter++;
        time(&si->last_used);
        sessionData[PrintSession::_duplicate(session)] = si;
      }
    else
      {
        LOG_SERVER << "Connection for user '" << save_user
                   << "' FAILED !!!" << endl;
      }
  }
```

Lorsqu'il s'agit de la réponse à un `connect`, le moniteur vérifie si la référence qui est renvoyée comme résultat est nulle. Dans ce cas, c'est que l'utilisateur n'est pas enregistré dans le serveur d'impression ou que le mot de passe qu'il a fourni est incorrect. La survenance d'un tel événement est affiché à l'écran.

Si par contre la référence du `PrintSession` qui est renvoyée par la méthode `connect` est non-nulle, le moniteur crée une structure de données qui est associée à cette référence. Cette opération lui permet de sauvegarder des informations relatives à la session, telles que le nom de l'utilisateur, l'heure de sa dernière utilisation ou l'état dans lequel il se situe.

Cette structure est récupérée au niveau du *RequestMonitor* lors de la réception de requêtes qui portent sur l'interface `PrintSession`. Les informations qui sont contenues dans celle-ci permettent notamment la vérification des transitions d'états et la mise en œuvre d'un *timeout* pour la session.

```
  if INTERFACE("IDL:PrintSession:1.0")
    {
      // Récupère les informations concernant la session
      SessionInfo* si = sessionData[ri.target];

      // Vérifie si la session est encore valide
      checkRequest_SessionTimeout(si);
```

L'extrait précédent provient de la partie du *RequestMonitor* qui s'occupe du monitoring des invocations sur l'interface `PrintSession`. On peut y voir l'invocation

de la méthode `checkRequest_SessionTimeout` qui pour but de vérifier la validité d'une session d'impression dans le temps. Celle-ci prend pour argument la structure qui a été créée lors du monitoring de la réponse et qui est retrouvée grâce au champ `target` du paramètre de la fonction `reply`. Le code C++ qui réalise cette vérification est reproduit ci-après.

```
void checkRequest_SessionTimeout(SessionInfo* si)
  throw(MonitorException)
{
  // Vérifie que la session n'est pas expirée... Si c'est le cas, on soulève une exception
  time_t current;
  time(&current);
  if ((current - si->last_used) > SESSION_TIMEOUT)
    {
      LOG_SESSION_REQUEST << "ERROR: Session timed out ("
                          << current - si->last_used << "s)! "
                          << "Aborting operation" << endl;
      throw MonitorException("Session timed out");
    }
  else time(&si->last_used);
}
```

A chaque utilisation d'une session d'impression, le temps écoulé depuis la dernière utilisation est calculé. Si celui-ci est supérieur à la constante `SESSION_TIMEOUT` une exception est soulevée et renvoyée à l'utilisateur par le biais du dispatcheur.

Résultat

Une capture d'écran permettant de visualiser le résultat du monitoring d'un *Print-Server* est reproduite à la figure 5.2

L'enregistrement du moniteur

Lors de la présentation de notre architecture, nous avons expliqué que les moniteurs doivent obligatoirement s'inscrire au niveau d'un *Dispatcher*.

Cette opération d'enregistrement est effectuée en appelant la méthode `registerMonitor` du moniteur qui souhaite s'inscrire sur le *Dispatcher* avec comme paramètres sa référence ainsi que la liste des interfaces qu'il supporte.

La référence vers le *Dispatcher* auquel est adressée cette demande, est obtenue au moyen du *Naming Service*. En ce qui concerne les moniteurs implémentés au moyen d'*ORBacus*/C++, une fonction qui permet de réaliser cette localisation est mise à la disposition des utilisateurs de notre implémentation sous le nom de `Dispatcher_impl::connectToDispatcher`.

```
zyk@Amok:~/Work/Memoire_Code/etude_de_cas
PrintServerMonitor, version 1.1
-------------------------------------------------------------------------------
Mon Sep  2 01:20:15 2002 PrintServer> User 'zyk' was successfully added to the server
Mon Sep  2 01:20:15 2002 PrintServer> User 'gav' was successfully added to the server
Mon Sep  2 01:20:15 2002 PrintServer> User 'zyk' was _not_ added to the server (already existing?)
Mon Sep  2 01:20:15 2002 PrintServer> Aborting user creation, password for user 'ged' is too short (size=3)
Mon Sep  2 01:20:15 2002 PrintServer> Connection for user 'zyk', referring as PrintSession[0]
Mon Sep  2 01:20:15 2002 PrintServer> Connection for user 'gav', referring as PrintSession[1]
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> Launching a print job
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> Finished printing
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Launching a print job
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Finished printing
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Launching a print job
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Finished printing
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> Launching a print job
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> Finished printing
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Launching a print job
Mon Sep  2 01:20:15 2002 PrintSession[1][gav]> Finished printing
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> WARNING: Reset while not printing!
Mon Sep  2 01:20:15 2002 PrintSession[0][zyk]> Disconnect
Mon Sep  2 01:21:08 2002 PrintSession[1][gav]> ERROR: Session timed out (53s)! Aborting operation
```

Fig. 5.2 – Résultat du monitoring d'un *PrintServer*

L'extrait ci-dessous provient de l'initialisation de notre intercepteur. On peut y observer la localisation du dispatcheur, la construction de la liste `ris` qui contient les interfaces à monitorer ainsi que l'invocation de la méthode `registerMonitor`.

```
// Crée l'implémentation du moniteur
RequestMonitor_impl* requestMonitor_impl = new RequestMonitor_impl;
// Active celle-ci
RequestMonitor_var requestMonitor = requestMonitor_impl->_this();

// Récupération de la référence du Dispatcher
Dispatcher_var dispatcher = Dispatcher_impl::connectToDispatcher(orb);

// Création de la liste des interfaces à monitorer
RepositoryIdSeq ris;
ris.length(2);
ris[0] = "IDL:PrintSession:1.0";
ris[1] = "IDL:PrintServer:1.0";

// Enregistrement du moniteur auprès du Dispatcher, en lui transmettant
// la liste des interfaces créées ci-dessus
dispatcher->registerMonitor(requestMonitor,ris);
```

Ceci clôture l'explication qui concerne la réalisation de moniteurs. Nous rappelons que l'exemple complet de celui qui a été utilisé tout au long de ce chapitre est disponible à l'annexe B.2.

5.4 Limitations du prototype

Le prototype que nous avons réalisé pour ce mémoire comporte quelques limitations qui ne sont pas liées à l'architecture mais plus précisément à une implémentation incomplète de celle-ci.

Absence de gestion des `send_other` et `send_exception`

L'interception des évènements `send_other` et `send_exception` n'est actuellement pas implémentée, que ce soit au niveau de l'intercepteur ou du dispatcheur.

Absence de possibilité de désenregistrement

Il n'est à ce jour pas encore possible de désenregistrer un moniteur ou un intercepteur au niveau du dispatcheur. Il faut donc redémarrer celui-ci chaque fois qu'un serveur ou un moniteur sont détruits.

Tolérance aux défaillances

Le prototype actuel est très sensible aux défaillances qui pourraient se produire dans un des composants. Ce problème est fortement lié au précédent : si un composant n'est plus disponible il devrait être retiré automatiquement du dispatcheur.

5.5 Évaluation des performances

Bien évidemment, le monitoring à un coût en termes de performances. Celui-ci est imputable à la lenteur des réseaux et au temps nécessaire pour l'analyse des requêtes. L'observation de cet impact fait l'objet de cette section.

5.5.1 Mesures

Nous avons essayé de mesurer l'augmentation du temps de réponse d'un composant suite au monitoring de celui-ci. Afin d'isoler les délais qui sont produits par l'architecture, nous avons employé des moniteurs qui n'effectuent aucune opération. Pour la même raison, le serveur utilisé pour les mesures est également très simple, il s'agit d'une fonction qui prend un entier en paramètre et qui renvoie celui-ci comme résultat.

Les mesures ont été effectuées à l'aide de deux machines reliées par un réseau fonctionnant à une vitesse de 100Mbit/seconde. La première machine est équipée d'un

Moniteurs	S,D,M	S/D,M	S,D/M	S/D/M
1	*4 ms*	*5 ms*	*6 ms*	*9 ms*
2	*7 ms*	*8 ms*	*10 ms*	*13 ms*
4	*13 ms*	*11 ms*	*18 ms*	*23 ms*
6	*19 ms*	*18 ms*	*29 ms*	*31 ms*
8	*22 ms*	*23 ms*	*35 ms*	*41 ms*

Tab. 5.2 – Délais d'invocation

processeur AMD Athlon XP 1800+ et dispose de 512Mb de mémoire centrale. Dans nos expériences il s'agit de la machine qui avait pour rôle d'exécuter les tâches qui nécessitaient la plus grande capacité de calcul. L'autre machine dispose d'un processeur AMD Athlon cadencé à 500Mhz, ainsi que 256Mb de mémoire.

Pour chacune des configurations que nous avons observées, nous avons connecté entre 1 et 8 moniteurs et calculé la moyenne des temps nécessaires pour monitorer 2000 requêtes et réponses. La somme de ces deux valeurs correspond au temps consacré au monitoring.

5.5.2 Résultats

Différentes configurations

Les différentes configurations et les résultats des mesures de délais sont reproduits dans le tableau 5.2. Nous avons utilisé les conventions suivantes pour dénommer les configurations de notre architecture : le serveur, le dispatcheur et les moniteurs sont respectivement représentés par les lettres **S**, **D** et **M**. Lorsque ces composants sont séparés par une virgule cela signifie qu'ils sont localisés sur une même machine. Ces groupements de composants peuvent également être séparés par un *slash*, celui-ci permet d'indiquer qu'ils sont localisés sur des machines différentes.

- **S,D,M** : Tous les composants sont sur une même machine. Le facteur qui limite les performances du système est la puissance de calcul du processeur car il n'y a aucune communication passant par le réseau.
- **S/D,M** : Le serveur est sur une machine, le dispatcheur ainsi que les moniteurs sont situées sur une autre machine. On remarque que les performances ne sont pas nettement inférieures à la configuration précédente, bien que la communication entre le serveur et le dispatcheur se fasse au travers du réseau.
- **S,D/M** : Cette fois ce sont le serveur et le dispatcheur qui se retrouvent sur une même machine et les moniteurs sont exécutés sur une machine distante. Lorsqu'il n'y a qu'un ou deux moniteurs, les performances de cette disposition sont proches de la précédente, mais l'écart grandit avec l'augmentation du

Moniteurs	S/D/M
1	*70 ms*
2	*72 ms*
4	*81 ms*
6	*95 ms*
8	*105 ms*

Tab. 5.4 – Délais d'invocation, moniteurs fictifs

nombre de moniteurs. Ceci confirme l'utilité du dispatcheur lorsque l'on est en présence de moyen de communication qui ont des vitesses différentes.

- **S/D/M** : Cette dernière configuration correspond à un serveur, un dispatcheur et des moniteurs qui sont tous situés sur des machines différentes. C'est également celle qui est la moins performante en raison du grand nombre de communications qui doivent transiter par le réseau. Néanmoins cette faiblesse est compensée par une plus grande flexibilité de l'architecture.

Nous avons également procédé à l'observation de la bande passante utilisée par les communications entre l'intercepteur, le dispatcheur et les moniteurs. A aucun moment nous n'avons constaté de trafic supérieur à 150Kb/s, ce qui est bien en dessous de la limite théorique du réseau. L'augmentation des délais qui se produit lors de l'accroissement du nombre de moniteurs est donc imputable à la vitesse du processeur.

Le *multithreading*

Pour vérifier l'utilité du *multithreading*, nous avons réalisé une mesure de performance en rajoutant dans les procédures `request` et `reply` des moniteurs, un appel vers la fonction `usleep` qui sert à provoquer une interruption de leur exécution pendant une durée exprimée en microsecondes. La valeur que nous avons utilisé est 25000, c'est-à-dire que nous provoquons un délais artificiel de 25 millisecondes au niveau du monitoring des requêtes et un délais identique au niveau des réponses.

Le tableau 5.4 reprend les différents délais mesurés au niveau de l'intercepteur. Ce que l'on peut constater c'est que le *multithreading* du dispatcheur a permis l'exécution en parallèle des moniteurs. Si celle-ci avait eu lieu de façon séquentielle, le temps nécessaire pour exécuter les 8 moniteurs aurait été d'au moins 400 millisecondes, sans compter les délais dus au réseau ainsi que ceux engendrés par notre application. Celui-ci est largement supérieur aux 105 millisecondes qui ont été observées au moyen de notre prototype.

Chapitre 6

Discussion

Notre discussion concernera en premier lieu l'évaluation et les possibilités d'amélioration de l'architecture de monitoring distribué qui vous a été présentée. Dans la seconde partie, nous parlerons succinctement des solutions alternatives que nous avions envisagées et nous expliquerons pourquoi celles-ci n'ont pas été retenues.

6.1 Évaluation et possibilités d'amélioration

Il ne nous a pas été possible d'effectuer une évaluation comparative de notre architecture car nous n'avons pas connaissance de plateformes de monitoring qui offrent les mêmes caractéristiques.

L'évaluation de notre architecture repose dès lors sur les trois objectifs principaux que nous nous sommes fixés lors de sa conception. Il s'agit de la facilité avec laquelle elle peut être intégrée dans un système distribué existant, de la capacité des moniteurs à modifier le déroulement d'une invocation et de son coût en terme de performances.

6.1.1 Les modifications nécessaires

La minimimisation des modifications qui doivent être apportées à une application existante afin d'intégrer notre système de monitoring faisait partie des objectifs principaux lors de la conception de notre architecture. Nous allons évaluer si celle-ci a pu être réalisée.

Modification de l'application

L'utilisation des intercepteurs CORBA a permis une intégration quasi transparente pour le serveur. En effet, grâce à eux, tous les échanges qui sont effectués à son

niveau sont observables sans que son comportement n'en soit modifié.

Cependant, comme nous l'avons vu au point 5.3.1, les intercepteurs CORBA ont besoin d'être initialisés avant l'ORB. Cette obligation implique la nécessité d'avoir accès au code source de l'application afin de pouvoir ajouter ces instructions d'initialisation. Elle est donc assez contraignante, surtout lorsqu'il s'agit de composants COTS (*Commercial Off-The-Shelf*) dont les sources ne sont pas mises à la disposition des clients.

Notons que cette limitation ne concerne pas tous les langages de programmation. Il est par exemple possible en Java de spécifier une classe contenant l'initialisation des intercepteurs à la ligne de commande.

Une autre contrainte liée à l'utilisation des intercepteurs réduit légèrement la transparence de notre architecture : il possible de devoir recompiler les interfaces IDL du serveur pour activer le support de ces intercepteurs. Cette opération ne s'avère nécessaire que lorsque les serveurs sont implémentés à l'aide d'*ORBacus*, les autres implémentations de CORBA générant par défaut des stubs et skeletons capables d'intercepter des requêtes.

Un choix que nous avons effectué lors de la conception de notre architecture était de ne pas apporter de modifications aux clients afin de faciliter la mise en place du monitoring. Malheureusement ceci a eu pour conséquence de ne pas disposer de renseignements relatifs à l'origine des requêtes au niveau de l'intercepteur du serveur. L'utilisation optionnelle d'intercepteurs au niveau des clients ainsi que celle des contextes de service permettraient d'étendre les fonctionnalités des intercepteurs sans obligatoirement porter atteinte à la facilité d'intégration.

Modification de l'infrastructure

En ce qui concerne l'architecture du système, le seul changement à apporter consiste dans l'ajout d'un *Naming Service*. Cette opération n'est nécessaire que si on ne l'utilisait pas auparavant. Il ne nous semble pas que ce soit une contrainte, étant donné que la majorité des applications distribuées font appel à ce mécanisme pour la localisation de leurs composants.

Notons également qu'il est tout à fait possible, en introduisant de légères modifications au prototype, de supporter d'autres services comme le *Trader Service* qui a été présenté au point 1.3.

6.1.2 L'interactivité des moniteurs

Une des caractéristiques principales de notre architecture est la possibilité pour les moniteurs de contrôler l'invocation des requêtes, notamment en leur permettant de soulever des exceptions.

Il n'est cependant pas possible pour un client de savoir pour quelle raison une requête a été avortée, car les messages qui sont associés à ces exceptions ne sont pas retransmis plus loin que les serveurs. Comme nous l'avons vu au premier chapitre, cette limitation est due aux intercepteurs CORBA qui ne peuvent que soulever des exceptions systèmes et des *ForwardException*s.

Une autre forme d'interactivité supportée par notre architecture consiste dans la possibilité pour les moniteurs de réaliser des appels sur d'autres objets distribués. Ceci permet aux moniteurs de remplir par exemple des fonctions de synchronisation.

Une amélioration de cette interactivité que nous avons envisagé, serait d'étendre les exceptions qu'un moniteur peut renvoyer afin d'inclure les *ForwardException*s, ce qui lui permettrait d'assurer une fonction de *load balancing*.

6.1.3 Performances

Il nous est difficile d'évaluer les performances de notre architecture puisque les applications que nous avons passées en revue au troisième chapitre n'ont pas de capacité d'interaction et que leur implémentation est réalisée dans un langage qui n'est pas réputé pour son efficacité.

Ce que l'on peut néanmoins constater c'est que l'introduction du dispatcheur entre les intercepteurs et les moniteurs a eu un effet positif sur les performances lorsque le comportement d'un serveur est vérifié par plusieurs moniteurs. Dans le tableau 5.2 cette amélioration peut être observée par la différence de délais entre les configurations **S/D,M** et **S,D/M**.

Comme nous l'avons mis en évidence dans la seconde série de mesures, le *multitheading* au niveau de ce dispatcheur permet également de réduire de manière significative le temps nécessaire au monitoring lorsque celui-ci est effectué par plusieurs moniteurs.

Dans les cas où il n'y a qu'un moniteur pour un serveur, le dispatcheur perd de son utilité et ralentit le système. Il devrait alors être possible pour un intercepteur de court-circuiter le dispatcheur en envoyant les informations interceptées directement à un moniteur. Notre architecture ne le permet pas encore, mais l'ajout de cette fonctionnalité devrait être facilitée par l'utilisation au niveau du dispatcheur et des moniteurs, de méthodes qui ont des signatures identiques (request,reply).

Il reste également une autre situation dans laquelle notre architecture engendre des délais superflus : il s'agit du cas d'un moniteur qui n'a pour fonction que de vérifier un sous-ensemble des opérations définies dans une interface. En effet, le critère actuellement utilisé par l'intercepteur et le dispatcheur afin de déterminer si une requête doit être retransmise aux moniteurs est le nom de l'interface sur laquelle porte celle-ci. Ce manque de précision a pour conséquence l'envoi d'informations inutiles depuis l'intercepteur vers les moniteurs et un impact négatif sur les performances générales du monitoring.

Cette limitation pourrait être levée facilement grâce à l'utilisation conjointe du nom de l'opération et de l'interface pour décider de l'envoi d'une requête au niveau des intercepteurs. Il en irait de même pour le dispatcheur ce qui lui permettrait de redistribuer de manière plus efficace les informations aux moniteurs.

Ce changement de critère de sélection impliquerait une légère modification des structures de données internes des intercepteurs et des moniteurs, ainsi que l'extension des messages échangés lors de l'inscription d'un moniteur ou d'un intercepteur.

6.2 Solutions alternatives

Nous avons choisi d'utiliser une approche qui utilise des moniteurs distribués, mais il ne s'agit pas de la seule possibilité. Même si le procédé d'interception utilisant les intercepteurs portables est supérieur aux autres, il aurait été possible de donner à ces intercepteurs plus d'autonomie. Dans ce cas, les moniteurs auraient été intégrés dans les intercepteurs, éliminant ainsi un grand nombre de communications. Cette solution a été envisagée au début du développement de l'architecture, mais trois raisons nous ont poussé à la rejeter :

- Pour des applications qui exigent des ressources importantes, la vérification des requêtes d'un objet aurait pu avoir un impact considérable sur les performances d'autres applications qui résident sur la même machine.
- Lorsque plusieurs objets collaborent entre-eux, il est nécessaire que chaque moniteur soit informé des événements qui ont eu lieu pour les autres objets. Ceci nécessite une synchronisation entre les différents intercepteurs qui peut s'avérer être une tâche complexe lorsqu'elle est réalisée sans l'aide d'un objet externe.
- Comme nous l'avons exposé lors de la présentation de notre architecture, l'implémentation des intercepteurs dépend directement de celle du serveur ainsi que de l'ORB qui est utilisé pour celui-ci. Dans le cas de systèmes hétérogènes cela impliquerait de devoir adapter, si pas réécrire, chaque moniteur selon l'architecture sur laquelle est situé le serveur.

Une autre approche qui aurait pu être envisagée est l'utilisation d'un service de distribution d'événements tel que l'*Event Service* [16] ou le *Notification Service* [15]. Ces solutions ont été écartées pour les raisons suivantes : d'une part les performances théoriques étaient inférieures en raison d'une plus grande complexité, et d'autre part ces services reposent sur des appels asynchrones qui sont à sens unique et théoriquement non-fiables.

Conclusion

L'architecture présentée dans ce mémoire répond correctement à nos objectifs. Elle permet à un ensemble de moniteurs d'accéder de manière transparente aux informations qui composent les requêtes adressées à un serveur, tout en exigeant un minimum de modifications.

Pour atteindre ces objectifs nous avons dû mettre au point un processus d'interception des requêtes, ainsi qu'une architecture permettant de les redistribuer à un ensemble de moniteurs. Des différentes approches que nous avons étudiées dans le second chapitre, une technique nous est apparue comme étant supérieure aux autres : il s'agit des *Portable Interceptors* qui font depuis peu partie des spécifications de CORBA.

Autour de ceux-ci nous avons construit une architecture qui est constituée de divers composants : un *accesseur* qui permet aux objets distants de configurer l'intercepteur, un dispatcheur qui redistribue des événements et des moniteurs qui vérifient les requêtes.

Ce qui différencie entre autres cette architecture des outils disponibles c'est sa capacité d'empêcher une invocation de parvenir à un serveur ce qui permet de le protéger de mauvaises manipulations. Il en est de même pour les résultats dont l'envoi peut être annulé par les moniteurs s'ils sont incorrects.

Perspectives

L'architecture présentée dans ce mémoire pourrait servir de base à d'autres projets dont l'objectif serait de l'améliorer et d'étendre ses fonctionnalités.

En effet nous avons relevé lors de son évaluation quelques limitations telles que le manque d'informations concernant l'origine d'une invocation ou l'impossibilité pour les moniteurs de faire parvenir des messages aux clients. Il serait de même possible d'améliorer les performances de notre architecture en rendant optionnelle l'utilisation du dispatcheur.

Il nous paraîtrait également possible de développer des moniteurs qui ne soient pas spécifiques à une application particulière, à l'opposé de celui que nous avons développé pour le serveur d'impression.

Une fonction qui pourrait être remplie par ces moniteurs serait la visualisation des interactions d'un système distribué, comme les applications que nous avons présentées dans le troisième chapitre de ce travail. La mise en œuvre de cette fonctionnalité nécessiterait au préalable l'ajout d'un mécanisme qui permettrait d'obtenir des informations concernant le client.

Il serait également envisageable de réaliser un moniteur qui fasse appel à l'outil de vérification ASAX (*the Advanced Security audit trail Analyser on uniX*) [7, 14]. Il s'agit d'un programme développé aux Facultés Universitaires Notre-Dame de la Paix qui a pour fonction de vérifier des flux de données, dans notre cas il s'agirait des différents événements retransmis par les intercepteurs. L'avantage de cette intégration est qu'il serait possible de réaliser des moniteurs en RUSSEL, le langage qu'utilise ASAX et qui a été spécialement conçu pour ce type d'application.

Dans ces perspectives, nous espérons donc vivement que notre travail sera utilisé par d'autres.

Bibliographie

[1] Fintan Bolton. *Pure CORBA*. Sams Publishing, July 2001. ISBN : 0-672-321812-1.

[2] N. Diakov, Marten van Sinderen, and Dick Quartel. *Monitoring extensions for component-based distributed software*, October 2000.

[3] Nikolay K. Diakov, Harold J. Batteram, Hans Zandbelt, and Marten van Sinderen. *Design and Implementation of a Framework for Monitoring Distributed Component Interactions*. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 227–240, 2000.

[4] Nikolay K. Diakov, Harold J. Batteram, Hans Zandbelt, and Marten van Sinderen. *Monitoring of Distributed Component Interactions*, 2000.

[5] Johan Moe Ericsson. *Understanding Distributed Systems via Execution Trace Data*.

[6] Jean-Marc Geib et Christophe Gransart et Philippe Merle. *CORBA : des concepts à la pratique*. Dunod, Paris, France, seconde edition, Octobre 1999. ISBN : 2-10-004806-6.

[7] Naji Habra, Baudouin Le Charlier, Aziz Mounji, and Isabelle Mathieu. *ASAX : Software Architecture and Rule-based Language for Universal Audit Trail Analysis*. In *Proceedings of the Second ESORICS*, volume 648, November 1992.

[8] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. *The Case for Reflective Middleware*. In *Communications of the ACM*, volume 45, June 2002.

[9] X. Logean, F. Dietrich, and S. Koppenhoefer. *Run-time monitoring of distributed applications*. In *Proceedings of Middleware'98*, September 1998.

[10] David L. Mills. *Network Time Protocol (NTP) Version 3*. Internet RFC, RFC 1305, March 1992.

[11] David L. Mills. *Simple Network Time Protocol (SNTP) Version 4*. Internet RFC, RFC 2030, October 1996.

[12] Johan Moe and David A. Carr. *Understanding distributed systems via execution trace data*. In *The proceedings of the ninth international workshop on program comprehension*, 2001.

[13] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, L.A. Tewksbury, and V. Kalogeraki. *The Eternal System : an Architecture for Enterprise Applications*. In

Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), 1999.

[14] Abdelaziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* Doctor of science, Facultés Universitaires Notre-Dame de la Paix, Namur (Belgium), September 1997.

[15] Object Management Group (OMG). *Notification Service Specification.* Technical Report Version 1.8, Object Management Group (OMG), June 2000.

[16] Object Management Group (OMG). *Event Service Specification.* Technical Report Version 1.1, Object Management Group (OMG), March 2001.

[17] Object Management Group (OMG). *Transaction Service Specification.* Technical Report Version 1.2, Object Management Group (OMG), September 2001.

[18] Object Management Group (OMG). *Security Service Specification.* Technical Report Version 1.8, Object Management Group (OMG), March 2002.

[19] Object Management Group (OMG). *The Common Object Request Broker : Architecture and Specification*, August 2002.

[20] Todd Scallan. *Monitoring and Diagnosis of CORBA Systems.* In *Java Developers Journal*, pages 138–144, June 2000.

[21] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented software architecture*, volume 2. John Wiley & Sons, September 2000. ISBN : 0471606952.

[22] Beth A. Schroeder. *On-Line Monitoring : A Tutorial. Computer*, 28(6) :72–78, June 1995.

[23] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt, and Ossama Othman. *The Design and Performance of Meta-Programming Mechanisms for Request Broker Middleware.* In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[24] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, and Bart Nieuwenhuis. *ORB Instrumentation for Management of CORBA.* In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).*

[25] Maarten Wegdam, Dirk-Jaap Plas, Aart van Halteren, and Bart Nieuwenhuis. *Using Message Reflection in a Management Architecture for CORBA.* In *DSOM*, pages 230–242, 2000.

Annexe A

Code source

A.1 ORBInitializer

Listing A.1 – ORBInitializer_impl.cpp

```
#include <stdlib.h>
#include <stdio.h>

#include <ORBInitializer_impl.h>

void
ORBInitializer_impl::pre_init(PortableInterceptor::ORBInitInfo_ptr info)
{
  cout << "- ORBInitializer_impl::pre_init(...)" << endl;
}

void
ORBInitializer_impl::post_init(PortableInterceptor::ORBInitInfo_ptr info)
{
  cout << "- ORBInitializer_impl::post_init(...)" << endl;

  // Enregistre l'intercepteur
  cout << "   - Creating & registering the interceptor" << endl;
  serverRequestInterceptor = new ServerRequestInterceptor_impl();
  try
    {
      info->add_server_request_interceptor(serverRequestInterceptor);
    }
  catch(const PortableInterceptor::ORBInitInfo::DuplicateName&)
    {
      assert(false);
    }
}

CORBA::ORB_ptr
ORBInitializer_impl::ORB_init(int argc,
                              char **argv)
{
  cout << "- ORBInitializer_impl::ORB_init(void)" << endl;

  cout << "   - Creation of the ORBInitializer" << endl;
  ORBInitializer_impl* initializer = new ORBInitializer_impl();

  cout << "   - Registering the ORBInitializer" << endl;
  PortableInterceptor::register_orb_initializer( initializer );
```

```
  cout << "   - Initialize the ORB" << endl;
  CORBA::ORB_ptr orb;
  try
    {
      orb = CORBA::ORB_init(argc, argv);
    }
  catch(const CORBA::Exception& ex)
    {
      cerr << "! Could not initialize the ORB" << endl;
      cerr << ex << endl;
      exit(1);
    }

  cout << "   - Finalizing the construction of the ServerRequestInterceptor" << endl;
   initializer->serverRequestInterceptor->finalize(orb);

  return orb;
}
```

Listing A.2 – ORBInitializer_impl.h

```
#ifndef __ORBINITIALIZER_IMPL_H__
#define __ORBINITIALIZER_IMPL_H__

#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <OB/PortableInterceptor.h>
#include <ServerRequestInterceptor_impl.h>

class ORBInitializer_impl : public PortableInterceptor::ORBInitializer,
                            public OBCORBA::RefCountLocalObject
{

  ServerRequestInterceptor_impl* serverRequestInterceptor;

public:
  virtual void pre_init(PortableInterceptor::ORBInitInfo_ptr info);
  virtual void post_init(PortableInterceptor::ORBInitInfo_ptr info);

  static CORBA::ORB_ptr ORB_init(int argc,
                                 char **argv);
};

#endif
```

A.2 ServerRequestInterceptor

Listing A.3 – ServerRequestInterceptor_impl.cpp

```cpp
#include <stdlib.h>
#include <stdio.h>

// Pour la fonction gettimeofday()
#include <sys/time.h>
#include <time.h>

#include <ORBInitializer_impl.h>
#include <ServerRequestInterceptor_impl.h>
#include <Accessor_impl.h>

#define BENCHMARK

#ifdef BENCHMARK
#include <limits.h>
#endif // BENCHMARK

// FIXME : Cette procédure suppose le POA connu...
CORBA::Object_var
getTarget(PortableInterceptor::ServerRequestInfo_ptr ri,
          PortableServer::POA_var poa)
{
  // Conversion de l'OctetSeq en ObjectId
  CORBA::OctetSeq_var oid_seq = ri->object_id();
  PortableServer::ObjectId oid(oid_seq->length(),
                               oid_seq->length(),
                               oid_seq->get_buffer(),
                               false);

  // Récupère une référence vers l'objet cible
  assert(!is_nil(poa));
  CORBA::Object_var target = poa->id_to_reference(oid);

  // Renvoi de celle-ci
  return target;
}


ServerRequestInterceptor_impl::ServerRequestInterceptor_impl(void) :
  dispatcher(Dispatcher::_nil())
{
  cout << "- ServerRequestInterceptor_impl::ServerRequestInterceptor_impl()" << endl;
}

char*
ServerRequestInterceptor_impl::name(void)
{
  return CORBA::string_dup("MonitoringInterceptor");
}


void
ServerRequestInterceptor_impl::destroy(void)
{
  cout << "- ServerRequestInterceptor_impl::destroy" << endl;
}


void
ServerRequestInterceptor_impl::receive_request_service_contexts(PortableInterceptor::ServerRequestInfo_ptr ri)
  throw (PortableInterceptor::ForwardRequest,
         CORBA::SystemException)
{
```

```
  // On ne fait rien
}


void
ServerRequestInterceptor_impl::receive_request(PortableInterceptor::ServerRequestInfo_ptr sri)
  throw (PortableInterceptor::ForwardRequest,
         CORBA::SystemException)
{
#ifdef BENCHMARK
  static unsigned long receive_usecs;
  static unsigned long receive_min_usec = ULONG_MAX;
  static unsigned long receive_calls;
#endif //BENCHMARK

  // On ne monitore pas l'accès aux accesseurs
  if (sri->target_is_a("IDL:Accessor:1.0")) return;

  cout << "> Receive request (" << sri->target_most_derived_interface() << "->" << sri->operation() << ")
       ... ";

  // Teste pour voir si "*" n'est pas dans la liste des interfaces... Si c'est le cas on monitore toutes les interfaces...
  set<string>::iterator i = repositoryIdSet.find("*");
  if (i == repositoryIdSet.end())
    {
      // Test pour voir si l'interface est présente dans la liste des interfaces à monitorer
      i = repositoryIdSet.find(sri->target_most_derived_interface());

      // Si ce n'est pas le cas, on continue l'exécution de la requête sans monitoring
      if (i == repositoryIdSet.end()) {
        cout << endl;
        return;
      }
    }

  // Alloue un requestInformations pour cette requête...
  RequestInformations *requestInformations = new RequestInformations;

  // Remplit les différents champs de la structure
  requestInformations->interceptorId = interceptorId;
  requestInformations->requestId = sri->request_id();
  requestInformations->repositoryId = sri->target_most_derived_interface();
  requestInformations->operation = sri->operation();
  requestInformations->target = getTarget(sri,rootPOA);
  requestInformations->expectAnswer = sri->response_expected();
  requestInformations->parameters = *(sri->arguments());
  struct timeval tv;
  gettimeofday(&tv,NULL);
  requestInformations->timeStamp.sec = tv.tv_sec;
  requestInformations->timeStamp.usec = tv.tv_usec;

  // Rajoute un pointeur dans le map requestId->requestInformations, ce qui permet de voir si il
  // s'agit d'un retour de fonction à monitorer plus tard, ainsi que pour avoir des informations
  // qui ne sont disponibles que dans ce point. Evidemment, cette opération n'est effectuée que s'il
  // ne s'agit pas d'un appel asynchrone.
  if (requestInformations->expectAnswer) requestIdMap[requestInformations->requestId] =
      requestInformations;

  try
    {
      dispatcher->request(*requestInformations);
    }
  catch (MonitorException& ex)
    {
      cerr << "Got a MonitorException, message = " << ex.reason << endl;
      throw CORBA::UNKNOWN(0, CORBA::COMPLETED_NO);
    }
```

```
  // Juste avant de passer la main au servant, nous sauvons le temps courant dans
  // le requestInformations
  gettimeofday(&tv,NULL);

#ifdef BENCHMARK
  // Calcul du temps écoulé...
  unsigned long sec = tv.tv_sec - requestInformations->timeStamp.sec;
  signed long usec = tv.tv_usec - requestInformations->timeStamp.usec;
  if (usec<0) {
    sec--;
    usec+=1000000;
  }
  receive_min_usec=(usec<receive_min_usec)?usec:receive_min_usec;
  receive_usecs+=usec;
  receive_calls++;

  cout << "MONITORED [" << sec << "s, " << (float)usec/1000.0f << "ms]"
       << " [AVG: " << (float)(receive_usecs/receive_calls)/1000.0f << "ms]"
       << " [MIN: " << (float)receive_min_usec/1000.0f << "ms]" << endl;
#endif //BENCHMARK

  requestInformations->timeStamp.sec = tv.tv_sec;
  requestInformations->timeStamp.usec = tv.tv_usec;
}


void
ServerRequestInterceptor_impl::send_reply(PortableInterceptor::ServerRequestInfo_ptr sri)
  throw (CORBA::SystemException)
{
#ifdef BENCHMARK
  static unsigned long send_usecs;
  static unsigned long send_min_usec = ULONG_MAX;
  static unsigned long send_calls;
#endif //BENCHMARK

  // La première chose à faire c'est sauver le temps courant...
  struct timeval tv;
  gettimeofday(&tv,NULL);

  RequestInformations* ri = NULL;
  map<unsigned long,RequestInformations*>::iterator i = requestIdMap.find(sri->request_id());
  if (i == requestIdMap.end()) return;
  ri = i->second;
  requestIdMap.erase(i);

  cout << "> Send reply (" << ri->repositoryId << "->" << ri->operation << ") ... ";

  ReplyInformations* replyInformations = new ReplyInformations;
  replyInformations->interceptorId = interceptorId;
  replyInformations->requestId = ri->requestId;
  replyInformations->repositoryId = ri->repositoryId;
  replyInformations->operation = ri->operation;
  replyInformations->target = ri->target;
  replyInformations->result = *(sri->result());

  // Calcul du temps écoulé pour l'exécution de la méthode
  unsigned long sec = tv.tv_sec - ri->timeStamp.sec;
  signed long usec = tv.tv_usec - ri->timeStamp.usec;
  if (usec<0) {
    sec--;
    usec+=1000000;
  }
  replyInformations->elapsedTime.sec = sec;
  replyInformations->elapsedTime.usec = usec;

  try
    {
```

```
      dispatcher->reply(*replyInformations);
    }
  catch (MonitorException& ex)
    {
      cerr << "Got a MonitorException, message = " << ex.reason << endl;
      throw CORBA::UNKNOWN(0,CORBA::COMPLETED_YES);
    }

#ifdef BENCHMARK
  struct timeval tv2;
  gettimeofday(&tv2,NULL);
  sec = tv2.tv_sec - tv.tv_sec;
  usec = tv2.tv_usec - tv.tv_usec;
  if (usec<0) {
    sec--;
    usec+=1000000;
  }
  send_min_usec=(usec<send_min_usec)?usec:send_min_usec;
  send_usecs+=usec;
  send_calls++;

  cout << "MONITORED [" << sec << "s, " << (float)usec/1000.0f << "ms]"
       << " [AVG: " << (float)(send_usecs/send_calls)/1000.0f << "ms]"
       << " [MIN: " << (float)send_min_usec/1000.0f << "ms]" << endl;
#endif //BENCHMARK

  delete ri;
  delete replyInformations;
}


void
ServerRequestInterceptor_impl::send_other(PortableInterceptor::ServerRequestInfo_ptr ri)
  throw(PortableInterceptor::ForwardRequest,
        CORBA::SystemException)
{
  cout << "> Send other (To be implemented)" << endl;
}


void
ServerRequestInterceptor_impl::send_exception(PortableInterceptor::ServerRequestInfo_ptr ri)
  throw(PortableInterceptor::ForwardRequest,
        CORBA::SystemException)
{
  cout << "> Send exception (To be implemented)" << endl;
}


bool
ServerRequestInterceptor_impl::finalize(CORBA::ORB_ptr orb)
{
  cout << "- ServerRequestInterceptor_impl::finalize(...)" << endl;

  dispatcher = Dispatcher_impl::connectToDispatcher(orb);

  // Résolution du RootPOA
  cout << "  - Resolving the RootPOA" << endl;
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
  rootPOA = PortableServer::POA::_narrow(obj);

  // Récupère la référence vers le POA manager, et activation de celui-ci
  cout << "  - Activate the POAManager" << endl;
  PortableServer::POAManager_var manager = rootPOA->the_POAManager();
  manager->activate();

  // Création d'un accesseur pour l'intercepteur
  cout << "  - Creating a new accessor for the interceptor" << endl;
```

```
  Accessor_impl* configImpl = new Accessor_impl(this);
  Accessor_var config = configImpl->_this();

  cout << "   - Registering the interceptor to the monitor" << endl;
  try
    {
      interceptorId = dispatcher->registerInterceptor(config);
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  return true;
}
```

Listing A.4 – ServerRequestInterceptor_impl.h

```
#ifndef __SERVERREQUESTINTERCEPTOR_IMPL_H__
#define __SERVERREQUESTINTERCEPTOR_IMPL_H__

#include <OB/CORBA.h>
#include <OB/PortableInterceptor.h>
#include <Dispatcher_impl.h>
#include <set>

class ServerRequestInterceptor_impl : public PortableInterceptor::ServerRequestInterceptor,
                                      public OBCORBA::RefCountLocalObject
{
  friend class Accessor_impl;

  // Référence vers le RootPOA pour le lookup du Target
  PortableServer::POA_var rootPOA;

  // Reference to the dispatcher
  Dispatcher_var dispatcher;

  set<unsigned long> requestIdSet;
  set<string> repositoryIdSet;

  map<unsigned long,RequestInformations*> requestIdMap;

  unsigned long interceptorId;

public:

  ServerRequestInterceptor_impl(void);

  virtual char* name(void);
  virtual void destroy(void);

  // At this interception point, Interceptors must get their service
  // context information from the incoming request and transfer it
  // to a slot of PortableInterceptor : :Current.
  virtual void receive_request_service_contexts(PortableInterceptor::ServerRequestInfo_ptr)
    throw (PortableInterceptor::ForwardRequest,
           CORBA::SystemException);

  // This interception point allows an Interceptor to query request
  // information after all the information, including operation
  // parameters, are available.
  virtual void receive_request(PortableInterceptor::ServerRequestInfo_ptr)
    throw (PortableInterceptor::ForwardRequest,
           CORBA::SystemException);

  // This interception point allows an Interceptor to query reply
  // information and modify the reply service context after the
```

```
  // target operation has been invoked and before the reply is
  // returned to the client.
  virtual void send_reply(PortableInterceptor::ServerRequestInfo_ptr)
    throw (CORBA::SystemException);

  // This interception point allows an Interceptor to query the
  // exception information and modify the reply service context
  // before the exception is raised to the client.
  virtual void send_exception(PortableInterceptor::ServerRequestInfo_ptr)
    throw (PortableInterceptor::ForwardRequest,
           CORBA::SystemException);

  // This interception point allows an Interceptor to query
  // the information available when a request results in
  // something other thatn a normal reply or an exception.
  virtual void send_other(PortableInterceptor::ServerRequestInfo_ptr)
    throw (PortableInterceptor::ForwardRequest,
           CORBA::SystemException);

  bool finalize(CORBA::ORB_ptr orb);
};

#endif
```

A.3 Accessor

Listing A.5 – Accessor.idl

```
#include <OB/Types.idl>

typedef sequence<CORBA::RepositoryId> RepositoryIdSeq;

interface Accessor
{
  void addInterfaces(in RepositoryIdSeq ris);
  void removeInterfaces(in RepositoryIdSeq ris);

  void print(in string s);
};
```

Listing A.6 – Accessor_impl.cpp

```
#include <OB/CORBA.h>
#include <Accessor_impl.h>

Accessor_impl::Accessor_impl(ServerRequestInterceptor_impl *sri)
  : interceptor(sri)
{
}

void Accessor_impl::addInterfaces(const RepositoryIdSeq &ris)
  throw (CORBA::SystemException)
{
  cout << "InterfaceConfiguration_impl::addInterfaces(...)" << endl;
  for (int i = 0; i < ris.length(); i++)
    {
      cout << "    " << ris[i] << endl;
      interceptor->repositoryIdSet.insert(ris[i]);
    }
}

void Accessor_impl::removeInterfaces(const RepositoryIdSeq &ris)
  throw (CORBA::SystemException)
{
  cout << "InterfaceConfiguration_impl::removeInterfaces(...)" << endl;
  for (int i = 0; i < ris.length(); i++)
    {
      cout << "    " << ris[i] << endl;
      set<string>::iterator j = interceptor->repositoryIdSet.find(ris[i]);
      if (j != interceptor->repositoryIdSet.end())
        {
          interceptor->repositoryIdSet.erase(j);
        }
    }
}

void Accessor_impl::print(const char* s)
  throw (CORBA::SystemException)
{
  cout << s << endl;
}
```

Listing A.7 – Accessor_impl.h

```
#ifndef ___Accessor_impl_h__
#define ___Accessor_impl_h__

#include <Accessor_skel.h>
#include <ServerRequestInterceptor_impl.h>
```

```
class ServerRequestInterceptor_impl;

class Accessor_impl : virtual public POA_Accessor,
                      virtual public PortableServer::RefCountServantBase
{
  ServerRequestInterceptor_impl *interceptor;

public:
  Accessor_impl(ServerRequestInterceptor_impl *sri);

  virtual void addInterfaces(const RepositoryIdSeq &ris)
    throw (CORBA::SystemException);

  virtual void removeInterfaces(const RepositoryIdSeq &ris)
    throw (CORBA::SystemException);

  virtual void print(const char* s)
    throw (CORBA::SystemException);
};

#endif
```

A.4 Dispatcher

Listing A.8 – Dispatcher.idl

```
#include <OB/Types.idl>
#include <OB/Dynamic.idl>

#include <Monitor.idl>

interface Dispatcher
{
  void request(in RequestInformations ri) raises (MonitorException);
  void reply(in ReplyInformations ri) raises (MonitorException);

  unsigned long registerInterceptor(in Accessor ic);
  void registerMonitor(in RequestMonitor mm, in RepositoryIdSeq ris);
};
```

Listing A.9 – Dispatcher_impl.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include <JTC/JTC.h>
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <Dispatcher_impl.h>

class DispatchThread : public JTCThread
{
  friend class Dispatcher_impl;

  typedef enum { REQUEST,REPLY } Operation;
  Operation operation;

  RequestMonitor_ptr requestMonitor;
  ReplyMonitor_ptr replyMonitor;

  const RequestInformations *requestInformations;
  const ReplyInformations *replyInformations;

  MonitorException monitorException;
  bool exceptionRaised;

public:

  DispatchThread(RequestMonitor_ptr mm,
                 const RequestInformations *ri)
    : operation(REQUEST),
      requestMonitor(mm),
      exceptionRaised(false)
  {
    requestInformations = ri;
  }

  DispatchThread(ReplyMonitor_ptr rm,
                 const ReplyInformations *ri)
    : operation(REPLY),
      replyMonitor(rm),
      exceptionRaised(false)
  {
    replyInformations = ri;
  }

  virtual void run()
  {
    try
```

```
      {
        switch (operation)
          {
          case REQUEST: replyMonitor = requestMonitor->request(*requestInformations); break;
          case REPLY: replyMonitor->reply(*replyInformations); break;
          }
      }
    catch(const MonitorException& ex)
      {
        exceptionRaised = true;
        monitorException = ex;
      }
  }
};

void Dispatcher_impl::request(const RequestInformations& ri)
  throw (MonitorException,
         CORBA::SystemException)
{
  cout << "> Request received for operation: " << ri.operation << ", requestId=" << ri.requestId << endl;

  RequestMonitorSet rms;

  // Ajoute dans rms touts les moniteurs qui sont intéressés par tout...
  RequestMonitorSetMap_i result = requestMonitorSetMap.find("*");
  if ( result != requestMonitorSetMap.end())
    {
      rms.insert(result->second->begin(),result->second->end());
    }

  // Ajout des moniteurs spécifiques
  result = requestMonitorSetMap.find((char*)ri.repositoryId);
  if ( result != requestMonitorSetMap.end())
    {
      rms.insert(result->second->begin(),result->second->end());
    }

  // S'il n'y a rien dans rms à la fin des deux ajouts, c'est qu'il y a un problème...
  if (rms.size() == 0)
    {
      cerr << "WARNING: Received a monitoring request for an interface that has no associated
           monitors" << endl;
      return;
    }

  // Pour chacun des RequestMonitors qui sont contenus dans rms, on crée un thread et on dispatche !
  vector<JTCThreadHandle> threadPool;
  for (RequestMonitorSet_i i = rms.begin(); i != rms.end(); i++)
    {
      JTCThreadHandle requestDispatchThread = new DispatchThread(*i,&ri);
      threadPool.push_back(requestDispatchThread);
      requestDispatchThread->start();
    }


  bool exceptionRaised = false;
  string exceptionMessage;
  for (int i=0;i<threadPool.size();i++)
    {
      threadPool[i]->join();
      // t est une référence vers le i'ième thread
      DispatchThread* t = (DispatchThread*)threadPool[i].get();
      // Si une exception a été soulevée dans ce thread, on change le flag global pour la requête entière
      exceptionRaised |= t->exceptionRaised;
      // On concatène les messages d'erreur pour pouvoir renvoyer une exception contenant tous les messages
      if (t->exceptionRaised) exceptionMessage += string(t->monitorException.reason) + string("\n");
    }
```

```
  // On désire sauvegarder les ReplyMonitors renvoyés par les moniteurs que dans le cas où il n'y a pas eu
  // d'exceptions, et évidemment si on doit s'attendre à recevoire une réponse plus tard
  if (!exceptionRaised && ri.expectAnswer)
    {
      ReplyMonitorVector *replyMonitorVector = new ReplyMonitorVector();

      for (int i=0;i<threadPool.size();i++)
        {
          // t est une référence vers le i'ième thread
          DispatchThread* t = (DispatchThread*)threadPool[i].get();
          // On rajoute le ReplyMonitor qui a été renvoyé par le moniteur dans une liste
          replyMonitorVector->push_back(t->replyMonitor);
        }
      // La liste créée ci-dessus est sauvegardée dans une 'base de données' pour consultation ultérieure
      replyMonitorVectorMap[InterceptorAndRequestId(ri.interceptorId,ri.requestId)] = replyMonitorVector;
    }
  else
    {
      // Si une exception s'est produite, où qu'on est en présence d'une requête asynchrone, on détruit tous les
      // ReplyMonitors qui ont été retournés, c'est à dire pour touts les moniteurs qui n'ont pas renvoyé d'exception
      for (int i=0;i<threadPool.size();i++)
        {
          // t est une référence vers le i'ième thread
          DispatchThread* t = (DispatchThread*)threadPool[i].get();
          if (!t->exceptionRaised) CORBA::release(t->replyMonitor);
        }
      // Finalement on envoie l'exception à l'appelant
      throw(MonitorException(exceptionMessage.c_str()));
    }
}

void Dispatcher_impl::reply(const ReplyInformations& ri)
  throw (MonitorException,
         CORBA::SystemException)
{
  cout << "> Reply received for operation: " << ri.operation << ", requestId=" << ri.requestId << endl;

  // Recherche dans la 'base de données' qui contient les ReplyMonitors si
  ReplyMonitorVectorMap_i i = replyMonitorVectorMap.find(InterceptorAndRequestId(ri.interceptorId,ri.
       requestId));
  if (i == replyMonitorVectorMap.end()) {
    cerr << "ERROR: Got a reply for a request that was not received" << endl;
    return;
  }

  ReplyMonitorVector *rmv = i->second;
  replyMonitorVectorMap.erase(i);

  vector<JTCThreadHandle> threadPool;

  // Crée et lance un thread pour chaque replyMonitor que l'on a recu, à condition
  // qu'ils ne soient pas nil
  for (ReplyMonitorVector_i j = rmv->begin(); j != rmv->end(); j++)
    {
      if (!CORBA::is_nil(*j))
        {
          JTCThreadHandle replyDispatchThread = new DispatchThread(*j,&ri);
          threadPool.push_back(replyDispatchThread);
          replyDispatchThread->start();
        }
    }

  bool exceptionRaised = false;
  string exceptionMessage;

  // Attend que tous les threads soient finis
  for (int i=0;i<threadPool.size();i++)
    {
```

```
      threadPool[i]->join();

      // t est une référence vers le i'ième thread
      DispatchThread* t = (DispatchThread*)threadPool[i].get();
      // Si une exception a été soulevée dans ce thread, on change le flag global pour la requête entière
      exceptionRaised |= t->exceptionRaised;
      // On concatène les messages d'erreur pour pouvoir une exception contenant tous les messages
      if (t->exceptionRaised) exceptionMessage += string(t->monitorException.reason) + string("\n");

      // On détruit le ReplyMonitor qui était associé
      CORBA::release(t->replyMonitor);
    }

  // On détruit la liste qui contenait les ReplyMonitors
  delete rmv;

  // Si une exception a été soulevée par un des ReplyMonitors, on soulève une exception
  // afin d'en avertir le client
  if (exceptionRaised)
    {
      throw(MonitorException(exceptionMessage.c_str()));
    }
}

CORBA::ULong Dispatcher_impl::registerInterceptor(Accessor_ptr ic)
  throw (CORBA::SystemException)
{
  cout << "> Registering a new interceptor" << endl;

  // On rajoute l'accesseur de l'intercepteur dans l'ensemble des
  // accesseurs enregistrées
  accessorSet.insert(Accessor::_duplicate(ic));

  // On construit une séquence de RepositoryId qui sont à monitorer, pour
  // l'ensemble des moniteurs enregistrés auprès du moniteur
  if (requestMonitorSetMap.size() > 0)
    {

      RepositoryIdSeq* ris = new RepositoryIdSeq();
      ris->length(requestMonitorSetMap.size());
      int j = 0;
      for (RequestMonitorSetMap_i i = requestMonitorSetMap.begin(); i != requestMonitorSetMap.end(); i++)
        {
          (*ris)[j++] = i->first.c_str();
        }

      // Cette séquence est transmise à l'accesseur de l'intercepteur
      // qui vient de se rajouter
      try
        {
          ic->addInterfaces(*ris);
        }
      catch (const CORBA::Exception& ex)
        {
          cerr << ex << endl;
          assert(false);
        }

      // Une fois transmise, cette séquence peut être libérée
      delete ris;
    }

  // Incremente interceptorCounter, et renvoie
  return(interceptorCounter++);
}


void Dispatcher_impl::registerMonitor(RequestMonitor_ptr mm,
```

```
                                    const RepositoryIdSeq& ris)
  throw (CORBA::SystemException)
{
  cout << "> Registering a new monitoring module" << endl;

  // Première étape : avertir tous les intercepteurs qu'il y a
  // des nouveaux RepositoryId à monitorer
  for (AccessorSet_i p=accessorSet.begin(); p!=accessorSet.end(); p++)
    {
      try
        {
          (*p)->addInterfaces(ris);
        }
      catch (const CORBA::Exception& ex)
        {
          cerr << ex << endl;
          assert(false);
        }
    }

  // Ensuite, pour chaque RepositoryId qui est supporté par le module,
  // on rajoute le module à la liste des modules auxquels les requetes
  // doivent être retransmis
  for (int i=0; i<ris.length(); i++)
    {
      RequestMonitorSet *s = NULL;
      RequestMonitorSetMap_i result = requestMonitorSetMap.find(ris[i]);
      if ( result != requestMonitorSetMap.end())
        {
          s = result->second;
        }
      else
        {
          s = new RequestMonitorSet();
          requestMonitorSetMap[ris[i]] = s;
        }
      s->insert(RequestMonitor::_duplicate(mm));
    }
}

// Méthode statique qui permet à un moniteur de récupérer la référence
// vers le dispatcher au moyen du Naming Service
Dispatcher_ptr Dispatcher_impl::connectToDispatcher(CORBA::ORB_ptr orb)
{
  JTCInitialize initialize;
  Dispatcher_var dispatcher;
  CORBA::Object_var obj;

  // Résolution du Naming Service
  try
    {
      obj = orb->resolve_initial_references("NameService");
    }
  catch(const CORBA::ORB::InvalidName&)
    {
      cerr << "Can't resolve 'NameService'" << endl;
      assert(false);
    }
  assert (!CORBA::is_nil(obj));

  CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow(obj);
  assert (!CORBA::is_nil(nc));

  // Crée et binde un Naming Contexts
  CosNaming::Name monitorName;

  monitorName.length(1);
  monitorName[0].id = CORBA::string_dup("monitor_dispatcher");
```

```
  monitorName[0].kind = CORBA::string_dup("");

  try
    {
      obj = nc->resolve(monitorName);
      assert (!CORBA::is_nil(obj));
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  try
    {
      dispatcher = Dispatcher::_narrow(obj);
      assert (!CORBA::is_nil(dispatcher));
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  return dispatcher._retn();
}
```

Listing A.10 – Dispatcher_impl.h

```
#ifndef ___Dispatcher_impl_h__
#define ___Dispatcher_impl_h__

#include <Dispatcher_skel.h>
#include <vector>
#include <set>
#include <map>

class Dispatcher_impl : virtual public POA_Dispatcher,
                        virtual public PortableServer::RefCountServantBase
{
  typedef set<Accessor_ptr> AccessorSet;
  typedef AccessorSet::iterator AccessorSet_i;

  typedef set<RequestMonitor_ptr> RequestMonitorSet;
  typedef RequestMonitorSet::iterator RequestMonitorSet_i;

  typedef map<string,RequestMonitorSet*> RequestMonitorSetMap;
  typedef RequestMonitorSetMap::iterator RequestMonitorSetMap_i;

  typedef vector<ReplyMonitor_ptr> ReplyMonitorVector;
  typedef ReplyMonitorVector::iterator ReplyMonitorVector_i;

  typedef pair<CORBA::ULong,CORBA::ULong> InterceptorAndRequestId;
  typedef map<InterceptorAndRequestId,ReplyMonitorVector*> ReplyMonitorVectorMap;
  typedef ReplyMonitorVectorMap::iterator ReplyMonitorVectorMap_i;

  AccessorSet accessorSet;
  RequestMonitorSetMap requestMonitorSetMap;
  ReplyMonitorVectorMap replyMonitorVectorMap;
  CORBA::ULong interceptorCounter;

public:

  virtual void request(const RequestInformations &ri)
    throw (MonitorException,CORBA::SystemException);

  virtual void reply(const ReplyInformations &ri)
    throw (MonitorException,CORBA::SystemException);
```

```
  virtual CORBA::ULong registerInterceptor(Accessor_ptr ic)
    throw (CORBA::SystemException);

  virtual void registerMonitor(RequestMonitor_ptr mm,
                               const RepositoryIdSeq& ris)
    throw (CORBA::SystemException);

  static Dispatcher_ptr connectToDispatcher(CORBA::ORB_ptr orb);
};

#endif
```

Listing A.11 – Dispatcher_main.cpp

```
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include <Dispatcher_impl.h>

int main(int argc, char **argv)
{
  CORBA::ORB_var orb;

  try
    {
      cout << "- Initializing the ORB" << endl;
      orb = CORBA::ORB_init(argc, argv);
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  // Resolve Root POA
  cout << "- Resolving the RootPOA" << endl;
  CORBA::Object_var poaObj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(poaObj);

  // Get a reference to the POA manager
  PortableServer::POAManager_var manager = rootPOA->the_POAManager();

  // Create implementation object
  Dispatcher_impl *monitorImpl = new Dispatcher_impl;
  Dispatcher_var monitor = monitorImpl->_this();

  // Get naming service
  cout << "- Resolving the NameService" << endl;
  CORBA::Object_var obj;
  try
    {
      obj = orb->resolve_initial_references("NameService");
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }
  assert (!CORBA::is_nil(obj));

  CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow(obj);
  assert (!CORBA::is_nil(nc));

  try
    {
      // Create and bind some Naming Contexts
      cout << "- Binding the 'monitor'" << endl;
      CosNaming::Name monitorName;
```

```
      monitorName.length(1);
      monitorName[0].id = CORBA::string_dup("monitor_dispatcher");
      monitorName[0].kind = CORBA::string_dup("");
      nc->bind(monitorName,monitor);

      // Run implementation
      cout << "----> Running <----" << endl;
      manager->activate();
      orb->run();

      // Unregister names with the Naming Service
      nc->unbind(monitorName);
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  if (! CORBA::is_nil(orb))
    {
      try
        {
          orb->destroy();
        }
      catch (const CORBA::Exception& ex)
        {
          cerr << ex << endl;
          assert(false);
        }
    }
}
```

A.5 Monitor

Listing A.12 – Monitor.idl

```
#include <OB/Types.idl>
#include <OB/Dynamic.idl>
#include <Accessor.idl>

exception MonitorException
{
  string reason;
};

struct Time
{
  long sec;
  long usec;
};

struct RequestInformations
{
  unsigned long interceptorId;
  unsigned long requestId;
  CORBA::RepositoryId repositoryId;
  string operation;
  Object target;

  boolean expectAnswer;
  Dynamic::ParameterList parameters;
  Time timeStamp;
};

struct ReplyInformations
{
  unsigned long interceptorId;
  unsigned long requestId;
  CORBA::RepositoryId repositoryId;
  string operation;
  Object target;

  any result;
  Time elapsedTime;
};

interface ReplyMonitor
{
  void reply(in ReplyInformations ri)
    raises (MonitorException);
};

interface RequestMonitor
{
  ReplyMonitor request(in RequestInformations ri)
    raises (MonitorException);
};
```

Annexe B

Étude de cas : le serveur d'impression

B.1 PrintServer

Listing B.1 – PrintServer.idl

```
interface PrintSession
{
  oneway void reset();
  boolean print(in string data);
  void disconnect();
};

interface PrintServer
{
  boolean addUser(in string login, in string password);
  PrintSession connect(in string login , in string password);
};
```

Listing B.2 – PrintServer_impl.cpp

```
#include <unistd.h>
#include <OB/CORBA.h>
#include <PrintServer_impl.h>

void PrintSession_impl::reset()
  throw(CORBA::SystemException)
{
  cout << "SEND: Reset to printer" << endl;
}

CORBA::Boolean PrintSession_impl::print(const char* data)
  throw(CORBA::SystemException)
{
  cout << "Printing: " << data << endl;
  return true;
}

void PrintSession_impl::disconnect()
  throw(CORBA::SystemException)
{
  CORBA::release(_this());
}

CORBA::Boolean PrintServer_impl::addUser(const char* login,
                                         const char* password)
```

```
  throw(CORBA::SystemException)
{
  if (usersDB.find(login) == usersDB.end())
    {
      usersDB[login] = password;
      return true;
    }
  else return false;
}

PrintSession_ptr PrintServer_impl::connect(const char* login,
                                           const char* password)
  throw(CORBA::SystemException)
{
  if (usersDB[login] != password) return PrintSession::_nil();
  PrintSession_impl* psi = new PrintSession_impl;
  return psi->_this();
}
```

Listing B.3 – PrintServer_impl.h

```
#ifndef ___PrintServer_impl_h__
#define ___PrintServer_impl_h__

#include <PrintServer_skel.h>
#include <map>

class PrintSession_impl : virtual public POA_PrintSession,
                          virtual public PortableServer::RefCountServantBase
{
public:

  virtual void reset()
    throw(CORBA::SystemException);

  virtual CORBA::Boolean print(const char* data)
    throw(CORBA::SystemException);

  virtual void disconnect()
    throw(CORBA::SystemException);
};

class PrintServer_impl : virtual public POA_PrintServer,
                         virtual public PortableServer::RefCountServantBase
{
  typedef map<string,string> UsersMap;
  typedef UsersMap::iterator UsersMap_i;
  UsersMap usersDB;

public:

  virtual CORBA::Boolean addUser(const char* login,
                                 const char* password)
    throw(CORBA::SystemException);

  virtual PrintSession_ptr connect(const char* login,
                                   const char* password)
    throw(CORBA::SystemException);
};

#endif
```

Listing B.4 – PrintServer_main.cpp

```
#include <OB/CORBA.h>
#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
```

```
#include <errno.h>

#include <PrintServer_impl.h>
#include <ORBInitializer_impl.h>

int main(int argc, char* argv[])
{
  // Initialise l'intercepteur et l'ORB
  CORBA::ORB_var orb = ORBInitializer_impl::ORB_init(argc,argv);

  // Obtient une référence vers le RootPOA
  CORBA::Object_var poaObj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(poaObj);

  // Obtient une référence vers le POA manager
  PortableServer::POAManager_var manager = rootPOA->the_POAManager();

  // Création de l'objet d'implémentation
  PrintServer_impl* PrintServerImpl = new PrintServer_impl();

  // Activation de celui-ci
  PrintServer_var PrintServer = PrintServerImpl->_this();

  // Sauve la référence dans un fichier
  CORBA::String_var s = orb->object_to_string(PrintServer);
  const char* refFile = "PrintServer.ref";
  ofstream out(refFile);
  if(out.fail()) {
    cerr << argv[0] << ": can't open `" << refFile << "': "
         << strerror(errno) << endl;
    return EXIT_FAILURE;
  }
  out << s << endl;
  out.close();

  // Exécute l'implémentation
  manager->activate();
  orb->run();

  if (!CORBA::is_nil(orb))
    {
      try
        {
          orb->destroy();
        }
      catch(const CORBA::Exception& ex)
        {
          cerr << ex << endl;
          assert(false);
        }
    }

  return 0;
}
```

B.2 PrintServerMonitor

Listing B.5 – PrintServerMonitor.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#include <OB/CORBA.h>
#include <hash_map>
#include <PrintServerMonitor.h>
#include <Dispatcher_impl.h>
#include <PrintServer.h>

#define MONITOR_NAME "PrintServerMonitor"
#define MONITOR_VERSION "1.1"

#define MIN_PASSWORD_LENGTH 5
#define SESSION_TIMEOUT 30

struct eqobj
{
  bool operator()(const CORBA::Object_ptr o1, const CORBA::Object_ptr o2) const
  {
    return o1->_is_equivalent(o2);
  }
};

struct hashobj
{
  bool operator()(const CORBA::Object_ptr o) const
  {
    return o->_hash(LONG_MAX);
  }
};

typedef hash_map<CORBA::Object_ptr,SessionInfo*,hashobj,eqobj> SessionDataMap;
typedef SessionDataMap::iterator SessionDataMap_i;
SessionDataMap sessionData;

static unsigned int id_counter = 0;
static bool printing = false;

const char* getTime(void)
{
  time_t now;
  time(&now);
  char* result = ctime(&now);
  result[strlen(result)-1] = ' ';
  return result;
}

ReplyMonitor_impl::ReplyMonitor_impl()
  : save_user(NULL)
{
}

ReplyMonitor_impl::ReplyMonitor_impl(const char* user)
  : save_user(strdup(user))
{
}

ReplyMonitor_impl::ReplyMonitor_impl(SessionInfo* si)
  : save_session_info(si)
{
}

ReplyMonitor_impl::~ReplyMonitor_impl()
```

```
{
  if (save_user) delete save_user;
}


#define INTERFACE(x) (strcmp(ri.repositoryId,x)==0)
#define OPERATION(x) (strcmp(ri.operation,x)==0)

#define LOG_SERVER cout << getTime() << "PrintServer> "
#define LOG_SESSION_REQUEST cout << getTime() << "PrintSession[" \
                                 << si->id << "][" << si->user << "]> "
#define LOG_SESSION_REPLY cout << getTime() << "PrintSession[" \
                               << save_session_info->id << "][" \
                               << save_session_info->user << "]> "


ReplyMonitor_ptr checkRequest_PrintServer(const RequestInformations& ri)
  throw(MonitorException)
{
  // Extraction des paramètres de la requête
  const char* user;
  assert(ri.parameters[0].argument >>= user);
  const char* password;
  assert(ri.parameters[1].argument >>= password);

  if (OPERATION("addUser"))
    {
      // Vérifie la taille du mot de passe, soulève une exception
      // si celui-ci est trop court. Cette erreur est également
      // loggée.
      if (strlen(password) < MIN_PASSWORD_LENGTH)
        {
          LOG_SERVER << "Aborting user creation, password for user '"
                     << user << "' is too short "
                     << "(size=" << strlen(password) << ")" << endl;
          throw MonitorException("Password too short");
        }
    }

  // Nous désirons inscrire un moniteur qui va inspecter les réponses
  // pour les deux opérations du PrintServer
  ReplyMonitor_impl* rmi = new ReplyMonitor_impl(user);
  return rmi->_this();
}

void checkRequest_SessionTimeout(SessionInfo* si)
  throw(MonitorException)
{
  // Vérifie que la session n'est pas expirée... Si c'est le cas, on soulève une exception
  time_t current;
  time(&current);
  if ((current - si->last_used) > SESSION_TIMEOUT)
    {
      LOG_SESSION_REQUEST << "ERROR: Session timed out ("
                          << current - si->last_used << "s)! "
                          << "Aborting operation" << endl;
      throw MonitorException("Session timed out");
    }
  else time(&si->last_used);
}


ReplyMonitor_ptr RequestMonitor_impl::request(const RequestInformations& ri)
  throw(MonitorException,
        CORBA::SystemException)
{
  if INTERFACE("IDL:PrintServer:1.0")
    {
```

```
      return checkRequest_PrintServer(ri);
    }
  else
    if INTERFACE("IDL:PrintSession:1.0")
      {
        // Récupère les informations concernant la session
        SessionInfo* si = sessionData[ri.target];

        // Vérifie si la session est encore valide
        checkRequest_SessionTimeout(si);

        if (OPERATION("print"))
          {
            // Ne pas autoriser deux impressions simultanées pour la même session
            if (si->state != PRINTING)
              {
                LOG_SESSION_REQUEST « "Launching a print job" « endl;
                si->state = PRINTING;
              }
            else
              {
                LOG_SESSION_REQUEST « "Aborting a new print job,"
                                    « " alreading printing!" « endl;
                throw MonitorException("Already printing");
              }
          }
        else
          if (OPERATION("disconnect"))
            {
              // Si on demande à être déconnecté alors qu'on est en train
              // d'imprimer, une erreur est affichée
              if (si->state != IDLE)
                {
                  LOG_SESSION_REQUEST « "WARNING: Disconnect while printing!"
                                      « endl;
                }
            }
          else
            if (OPERATION("reset"))
              {
                // Si un reset parvient au moniteur alors qu'il n'y a pas
                // d'impression en cours, cette erreur est affichée à l'écran
                if (si->state != PRINTING)
                  {
                    LOG_SESSION_REQUEST « "WARNING: Reset while not printing!"
                                        « endl;
                  }
                else
                  {
                    LOG_SESSION_REQUEST « "Reset" « endl;
                  }
              }

        if (ri.expectAnswer)
          {
            ReplyMonitor_impl* rmi = new ReplyMonitor_impl(si);
            return rmi->_this();
          }
      }

  return ReplyMonitor::_nil();
}


void ReplyMonitor_impl::reply(const ReplyInformations& ri)
  throw(MonitorException,
        CORBA::SystemException)
{
```

```
  if OPERATION("connect")
    {
      PrintSession_ptr session;
      assert(ri.result >>= session);

      if (!CORBA::is_nil(session))
        {
          LOG_SERVER << "Connection for user '" << save_user << "'"
                     << ", referring as PrintSession[" << id_counter
                     << "]" << endl;
          SessionInfo *si = new SessionInfo;
          si->user = strdup(save_user);
          si->state = IDLE;
          si->id = id_counter++;
          time(&si->last_used);
          sessionData[PrintSession::_duplicate(session)] = si;
        }
      else
        {
          LOG_SERVER << "Connection for user '" << save_user
                     << "' FAILED !!!" << endl;
        }
    }
  else
    if OPERATION("addUser")
      {
        bool r;
        assert(ri.result >>= CORBA::Any::to_boolean(r));

        if (r) LOG_SERVER << "User '" << save_user
                          << "' was successfully added to the server" << endl;
        else LOG_SERVER << "User '" << save_user
                        << "' was _not_ added to the server"
                        << " (already existing?)" << endl;
      }
    else
      if OPERATION("print")
        {
          LOG_SESSION_REPLY << "Finished printing" << endl;
          save_session_info->state = IDLE;
        }
      else
        if OPERATION("disconnect")
          {
            LOG_SESSION_REPLY << "Disconnect" << endl;

            // Lorsque la déconnexion est terminée, on efface les informations
            // qui étaient associées à la session.
            SessionDataMap_i i = sessionData.find(ri.target);
            if (i != sessionData.end())
              {
                SessionInfo *si = (*i).second;
                delete si->user;
                delete si;
                sessionData.erase(i);
              }
          }
}


int main(int argc, char **argv)
{
  CORBA::ORB_var orb;

  // Initialisation de l'ORB
  try
    {
      orb = CORBA::ORB_init (argc, argv);
```

```
    }
  catch (const CORBA::Exception& ex)
    {
      cerr << ex << endl;
      assert(false);
    }

  // Résolution du RootPOA
  CORBA::Object_var poaObj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(poaObj);

  // Récupère la référence du POAManager
  PortableServer::POAManager_var manager = rootPOA->the_POAManager();
  manager->activate();

  // Crée l'implémentation du moniteur
  RequestMonitor_impl* requestMonitor_impl = new RequestMonitor_impl;
  // Active celle-ci
  RequestMonitor_var requestMonitor = requestMonitor_impl->_this();

  // Récupération de la référence du Dispatcher
  Dispatcher_var dispatcher = Dispatcher_impl::connectToDispatcher(orb);

  // Création de la liste des interfaces à monitorer
  RepositoryIdSeq ris;
  ris.length(2);
  ris[0] = "IDL:PrintSession:1.0";
  ris[1] = "IDL:PrintServer:1.0";

  // Enregistrement du moniteur auprès du Dispatcher, en lui transmettant
  // la liste des interfaces créées ci-dessus
  dispatcher->registerMonitor(requestMonitor,ris);

  cout << MONITOR_NAME << ", version " << MONITOR_VERSION << endl;
  cout << "---------------------------------------------------------------------------" << endl;

  // Exécution !
  orb->run();

  if (!CORBA::is_nil(orb))
    {
      try
        {
          orb->destroy();
        }
      catch (const CORBA::Exception& ex)
        {
          cerr << ex << endl;
          assert(false);
        }
    }
}
```

Listing B.6 – PrintServerMonitor.h

```
#ifndef ___RequestMonitor_impl_h__
#define ___RequestMonitor_impl_h__

#include <Monitor_skel.h>
#include <time.h>

typedef enum {IDLE,PRINTING} PrintSessionState;
typedef struct
{
  char *user;
  PrintSessionState state;
  unsigned int id;
  time_t last_used;
```

```
} SessionInfo;


class ReplyMonitor_impl : virtual public POA_ReplyMonitor,
                          virtual public PortableServer::RefCountServantBase
{
  char* save_user;
  CORBA::Object_ptr save_target;
  SessionInfo* save_session_info;

public:
  ReplyMonitor_impl();
  ReplyMonitor_impl(const char* user);
  ReplyMonitor_impl(SessionInfo* si);
  ~ReplyMonitor_impl();

  virtual void reply(const ReplyInformations& ri)
    throw(MonitorException,
          CORBA::SystemException);
};


class RequestMonitor_impl : virtual public POA_RequestMonitor,
                            virtual public PortableServer::RefCountServantBase
{
public:
  virtual ReplyMonitor_ptr request(const RequestInformations& ri)
    throw(MonitorException,
          CORBA::SystemException);
};

#endif
```