# Infrastructure logicielle multi-modèles pour l'accès à des servcies en mobilité

Aurelien Bocquet

N° d'ordre : 4330

Département de formation doctorale en informatique
UFR IEEA

Ecole Doctorale SPI Lille

# Infrastructure logicielle multi-modèles pour l'accès à des services en mobilité

## Thèse

présentée et soutenue publiquement le 1er décembre 2008
pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
spécialité Informatique

par

Aurélien BOCQUET

**Composition du jury :**

| | | |
|---|---|---|
| Président du jury : | M. Alain DERYCKE, Professeur | LIFL / Université de Lille I |
| Rapporteurs : | M. Guy BERNARD, Professeur | Telecom SudParis |
| | M. Laurent PAUTET, Professeur | ENST Paris |
| Examinateur : | M. Nikolaos GEORGANTAS, Chargé de recherche | INRIA - ARLES |
| Directeurs de thèse : | M. Jean-Marc GEIB, Professeur | LIFL / Université de Lille I |
| | M. Christophe GRANSART, Chargé de recherche | INRETS - LEOST |
| Invité : | M. Sylvain LECOMTE, Professeur | LAMIH / Université de Valenciennes |

# Informations administratives

**Titre de la thèse**
Infrastructure logicielle multi-modèles pour l'accès à des services en mobilité

**Université d'inscription**
Ecole Doctoral Sciences pour l'Ingénieur (ED-SPI)
Bâtiment P3, bureau 202, 59655 Villeneuve d'Ascq Cedex
+33 (0)3.20.43.67.06

Université des Sciences et Technologies de Lille (USTL)
Cité Scientifique, 59655 Villeneuve d'Ascq Cedex
+33 (0)3.20.43.43.43

**Laboratoire d'accueil**
Laboratoire Electronique, Ondes et Signaux pour les Transports (LEOST)
unité de recherche de l'Institut National de Recherche sur les Transports et leur Sécurité (INRETS)
20 rue Elisée Reclus, BP 317, 59666 Villeneuve d'Ascq Cedex
+33 (0)3.20.43.83.43

**Laboratoire académique partenaire**
Laboratoire d'Informatique Fondamentale de Lille (LIFL)
UMR 8022 USTL/CNRS
Bâtiment M3
59655 Villeneuve d'Ascq Cedex
+33 (0)3.28.77.85.41

**Encadrants**

DIRECTEUR DE THÈSE : Jean-Marc GEIB, professeur au LIFL, `jean-marc.geib@lifl.fr`
CO-ENCADRANT DE THÈSE : Christophe GRANSART, chargé de recherche au LEOST, `christophe.gransart@inrets.fr`

**Financement de thèse**

Co-financement INRETS / Région Nord-Pas-de-Calais

**Dates**

DÉBUT DES TRAVAUX : 1er septembre 2005
DÉBUT DU FINANCEMENT : 1er novembre 2005
SOUTENANCE DE LA THÈSE : 1er décembre 2008

# Remerciements

Tout d'abord, je tiens à remercier les membres de mon jury pour le temps et l'intérêt qu'ils ont portés à mon travail. En particulier, je remercie Laurent Pautet et Guy Bernard, pour avoir accepté le rôle de rapporteurs. Ils attestent ainsi de l'intérêt de cette thèse, ce mémoire et ces années de recherche.

Ces trois années de travail de thèse ont été financées par l'INRETS (Institut National de Recherche sur les Transports et leur Sécurité) et la région Nord-Pas de Calais. Je leur suis reconnaissant d'avoir cru en notre projet et de l'avoir ainsi soutenu.

Je tiens à exprimer toute ma gratitude au personnel du LEOST (Laboratoire Electronique, Ondes et Signaux pour les Transports) pour m'avoir accueilli durant ces années, et m'avoir donné les moyens de mener cette thèse à bien. Je remercie plus particulièrement Marion Berbineau, directrice de l'Unité de Recherche, ainsi que Lidwine Crampon et Olivier Delafraye au secrétariat, pour leur travail et leurs conseils administratifs qui m'ont permis de me centrer sur mon travail de recherche.

Certains travaux de cette thèse se sont inscrits dans le projet REVE. Je souhaite remercier tous les membres de ce projet, et en particulier Lionel Seinturier, responsable du projet, pour m'avoir permis d'y participer activement.

Julien Merlin et Franck Desbuquois, étudiants en Master 1 informatique, m'ont aidé dans mes recherches en établissant une batterie de tests d'évalua-

tion de mon travail. Un grand merci à eux pour la motivation et l'efficacité dont ils ont fait preuve.

La direction de ma thèse a été assurée par Jean-Marc Geib. Outre la confiance qu'il m'a portée pour mener ces travaux, je le remercie d'avoir pris le temps de me conseiller lorsque cela était nécessaire.
Je remercie également grandement Christophe Gransart, co-directeur de thèse, qui a su croire en moi, me conseiller et me guider avant et pendant ces trois années de recherche. Il est la preuve que l'on peut motiver sans contraindre.

Je remercie mes parents pour avoir su me soutenir et m'épauler, et en particulier mon père pour m'avoir donné le goût de la recherche. Enfin, je remercie Morgane, pour m'avoir ~~obligé~~ encouragé à persévérer lorsque la motivation déclinait, et Eliott, qui a su faire ses longues nuits assez rapidement pour me laisser travailler.

# Résumé

Les intergiciels sont aujourd'hui incontournables lorsqu'il s'agit de développer des applications réparties. Des simples Web Services aux architectures n-tiers, d'une unique communication client / serveur à un réseau dynamique pair-à-pair, chaque conception requiert des outils adaptés et performants. En complément de chaque utilisation spécifique des intergiciels, leur contexte de déploiement nécessite des mécanismes particuliers afin de s'adapter au mieux à la situation.

Face à ces besoins, les intergiciels proposent des modèles de programmation et de communication différents, fournissant des moyens de communication efficaces dans certaines situations.

La mobilité introduit une problématique supplémentaire pour ces intergiciels. D'une part l'interopérabilité devient inévitable ; le nombre de composants répartis susceptibles d'être utilisés en mobilité est immense, et les composants peuvent être développés avec différents intergiciels. D'autre part le contexte varie, et avec lui les conditions et capacités de communication évoluent.

Nous traitons dans cette thèse des impératifs actuels d'un intergiciel en mobilité. Nous proposons pour cela une approche multi-modèles, basée sur les travaux actuels dans ce domaine, et présentant des concepts novateurs.

Cette approche se compose d'un modèle de programmation générique, proposant différents types de communications synchrones, asynchrones, et basées

sur des patrons de conception. Elle se compose également d'une combinaison de modèles de communication, assurant l'interopérabilité avec les intergiciels standards, et offrant des possibilités de communications enrichies, capables de s'adapter aux changements de contextes.
Des politiques d'adaptation définissent les règles de combinaison des modèles en fonction d'observations du contexte, afin de se comporter au mieux face à ses évolutions.
Des mécanismes d'adaptation dynamique permettent à notre approche de proposer une prise en compte en temps réel des changements de contexte, et permettent également de reconfigurer le système pendant son exécution afin de répondre à des besoins de déploiement.

Nous avons validé notre approche au travers d'une application concrète aux problèmes engendrés par l'utilisation d'un proxy Internet à bord des trains : le développement d'un greffon multi-modèles a illustré et justifié notre approche, et l'évaluation de ce greffon a montré les bénéfices de celle-ci face aux changements de contexte.
Pour implémenter entièrement notre approche et proposer ainsi un intergiciel multi-modèles, nous avons conçu et développé notre infrastructure logicielle multi-modèles, proposant tous les concepts de l'approche. Une première version "statique" puis une version finale offrant les mécanismes d'adaptation dynamique ont été implémentées et permettent ainsi de profiter des bénéfices de notre approche multi-modèles.

*Mots-clés*

Intergiciel de communication, modèle de programmation, approche multi-modèles, annotations, adaptation au contexte, adaptation dynamique.

# Abstract

**Multi-model software infrastructure for the access to services in mobility.**

The middlewares are nowadays unavoidable when developing distributed applications. From simple Web Services to multitier architectures, from single client / server communication to a dynamic perr-to-peer network, every design needs adapted and efficient tools. In addition to every typical use of middlewares, their context of deployment needs special mechanisms in order to adapt to the situation.

In front of these needs, the middlewares offer different programming and communication models, supplying efficient ways to communicate in some situations.

The mobility introduces new problems for these middlewares. On one hand, interoperability becomes unavoidable ; the quantity of distributed components, which are likely to be used in mobility, is huge, and components may be designed with different middlewares. On the other hand, the context changes, and so do conditions and capabilities of communication.

In this thesis, we are dealing with the current requirements of a middleware in mobility. We thus propose a multi-model approach, based on the current works in this domain, and presenting innovative concepts.

This approach is composed by a generic programming model, which proposes different kinds of synchronous, asynchronous and design-pattern-based communications. It is also composed by a combination of communication models, ensuring interoperability with standard middlewares, and offering capabilities of enhanced communications, able to adapt to changes of context.
Adaptation policies define the rules of combination of models, regarding context observations, in order to behave the most efficient way, considering its evolution.
Dynamic adaptation mechanisms allow our approach to propose to handle context changes in realtime, and allow to reconfigure the system when it is running to answer to deployment needs.

Our approach has been validated through a concrete application to problems caused by the use of an embedded Internet proxy in trains : the design and development of a multi-model graft illustrated and justified our approach, and the evaluation of this graft demonstrated the benefits of this approach via-a-vis the changes of context.
To fully implement our approach and thus to propose a multi-model middleware, we designed and developed our multi-model software infrastructure, proposing all the concepts of the approach. A first "static" version, then a final one, which offers the mechanisms of dynamic adaptation, have been implemented and thus allow benefiting from our multi-model approach.

*Keywords*

Communication middleware, programming model, multi-model approach, annotations, adaptation to the context, dynamic adaptation.

# Table des matières

# Table des figures

# *Préface*

Aujourd'hui l'informatique, et en particulier la recherche informatique, se voit divisée en domaines d'expertise et axes de recherche (e.g. Programmation orientée objet, composants, aspects, systèmes embarqués, DSL, etc...) présentant chacun ses spécificités, et surtout son vocabulaire dédié.
Nous traitons dans cette thèse d'intergiciels de communication permettant la conception d'applications réparties. L'architecture de ces applications est basée sur la répartition de composants, mais le terme même de composant doit être interprété avec soin. En effet les composants, comme définis dans les modèles de composants tels que Fractal [BCL+04], vérifient des critères et propriétés précis [SGM02]. L'utilisation du terme de composant réparti dans ce document ne devra pas être comprise dans ce sens strict, mais simplement comme une partie d'application distribuée.

L'intitulé du sujet de cette thèse est resté identique depuis son commencement il y a plusieurs années. Certains projets connexes[1] existaient d'ores et déjà, mais la notion d'intergiciel "combiné" ou "générique" était récente. C'est pourquoi le vocabulaire employé alors n'était pas fixé : intergiciel schizophrène, interopérable, réflexif.
Le choix des termes du sujet de la thèse, et plus précisément la notion d'infrastructure "multi-modèles", s'inscrit dans l'optique de fournir un outil présentant différents types de communications, plutôt que plusieurs protocoles du même type. L'intérêt principal de notre proposition se situe d'ailleurs précisément dans cette nuance, comme nous l'exposerons par la suite. Les termes de "modèles de programmation et de communication" (termes utilisés pour décrire la commutation synchrone / asynchrone [BB02]) seront donc utilisés dans ce sens, et préférés aux termes de "personnalité applicative et protocolaire" (termes utilisés pour décrire PolyORB [VHPK04]) qui ne présentent pas, pour nous, la nuance suffisante pour décrire fidèlement notre approche.

[1] e.g. PolyORB et ReMMoC

# Introduction

L'infrastructure logicielle que nous proposons et présentons ici est tout d'abord née de besoins ressentis à plusieurs niveaux. Dans cette introduction, nous expliquerons dans un premier temps le contexte de cette thèse et des travaux qui en découlent. Puis nous exposerons la problématique principalement visée, pour enfin résumer notre proposition. Un aperçu de l'organisation de ce mémoire permettra d'en comprendre le déroulement.

## *1 Contexte des travaux*

Les travaux relatifs à cette thèse ont été menés au sein du LEOST[1], unité de recherche de l'INRETS[2]. De plus, ces recherches faisaient partie du projet JACQUARD[3], et font partie intégrante aujourd'hui du projet ADAM[4] de l'INRIA[5] et de l'équipe GOAL[6] du LIFL[7]. Les différentes problématiques visées par cette thèse lui permettent de pleinement s'intégrer à ces différents laboratoires et projets, soit par l'essence même de son sujet, soit par son domaine d'application.

[1] Laboratoire Electronique, Ondes et Signaux pour les Transports
[2] Institut National de Recherche sur les Transports et leur Sécurité
[3] Tissage de composants logiciels, projet arrêté le 31/12/2007
[4] Adaptive Distributed Applications and Middleware
[5] Institut National de Recherche en Informatique et en Automatique
[6] Groupe sur les Objets et composAnts Logiciels
[7] Laboratoire d'Informatique Fondamentale de Lille

### *1.1 Applications distribuées*

Les applications réparties ou distribuées représentent aujourd'hui la grande majorité des développements entrepris. Si le terme de répartition se comprenait autrefois au sens propre d'éloignement physique, la démocratisation des interactions entre applications et des architecture réparties (de type n-tiers) a rendu cette répartition omniprésente, même sur une seule et même plateforme d'exécution.

Le concept de répartition est essentiel, reconnu et accepté par les développeurs, mais soulève de nouvelles problématiques, liées d'une part aux différents types de communications utilisées, et d'autre part à l'hétérogénéité des applications. En effet, si la répartition permet d'utiliser des objets et composants répartis d'origines diverses, ceux-ci sont susceptibles de présenter des spécificités propres à chacun, et ces différences potentielles peuvent engendrer des problèmes de compatibilité et d'interopérabilité.

### *1.2 Domaine d'application : les transports*

Cette thèse est avant tout initiée, et financée en partie, par l'INRETS, qui axe ses recherches sur les transports. Dans ce domaine particulièrement les communications, et donc les intergiciels de communication ont une importance capitale dans la maintenance des applications existantes, et le développement de nouvelles applications. Ainsi ces intergiciels deviennent indispensables lorsqu'il s'agit de services aux personnes (comme le projet ICAU [RAG04]), de sécurité (comme le projet TESS [GARG02]), ou de système global de gestion des trains (projet IntegRAIL [CE05]). Les services et la sécurité dans les transports ont un besoin irrémédiable d'applications logicielles réparties, et celles-ci doivent pouvoir répondre aux nécessités particulières de ce domaine.

### *1.3 Programmation objet - vocabulaire utilisé*

Ces travaux se situent au niveau de la programmation objet, et non au niveau des composants comme beaucoup de travaux actuels. Nous l'avons voulu ainsi pour établir les bases saines et exploitables de notre approche et de notre architecture. En effet, la notion de composants apporte une abstraction supplémentaire pour le développeur, mais également des difficultés supplémentaires quant à la spécification et la réalisation d'un intergiciel. Néanmoins, toutes les théories et applications énoncées ici peuvent bien sûr être transposées au niveau composant ; nous l'avons d'ailleurs effectué en partie dans le cadre du projet REVE (chapitre V.1). Dans un second temps,

comme nous l'expliquerons plus en détails dans la conclusion, une intégration de notre approche dans un modèle de composants est visée.

## *2 Problématique initiale*

Cette thèse propose une infrastructure logicielle répondant à une problématique forte, et fréquemment pointée du doigt lors de développements d'applications réparties. Celles-ci doivent répondre à des exigences de plus en plus importantes en matière de rapidité de développement, de fiabilité, de performance, et d'interopérabilité. De plus, une même application peut être déployée dans des contextes totalement différents et contradictoires, ce qui complique d'autant plus son développement. Cette problématique est donc complexe, mais reflète les réels besoins actuels en matière de développement et de déploiement d'applications réparties.

### *2.1 Développement d'applications réparties*

Lorsqu'il s'agit de développer des applications réparties, le choix est vaste entre les différents moyens possibles d'y arriver. De nombreux modèles de communication existent (cf. chapitre I), et chacun offre ses avantages, inconvénients et limitations. Une fois celui-ci choisi, plusieurs modèles de programmation sont possibles pour l'utiliser ; ou vice-versa plusieurs modèles de communication sont possibles une fois le modèle de programmation choisi. Le modèle de programmation offre de nombreuses possibilités de communication entre objets ou composants, et également des services annexes (e.g. sécurité, persistance, transactions), mais demande avant tout de savoir l'utiliser : connaissance de l'API[1], marches à suivre pour l'initialisation (e.g. instanciation de l'ORB[2], gestion des erreurs et exceptions, ...

Les choix de modèles doivent être faits au début du développement, et un changement de ceux-ci nécessite une "refonte" de l'application, pour utiliser la nouvelle API. De plus, les modèles de programmation actuels pour applications réparties ne présentent guère plus d'un type de programmation, et un type de communication. Lorsqu'une application nécessite plusieurs de ceux-ci, il est nécessaire de combiner plusieurs modèles à la fois, ce qui demande un lourd travail de développement. Il en est de même lors d'évolutions d'applications existantes, et l'utilisation d'un nouveau modèle de programmation ou

[1] Application Programming Interface, interface de programmation définissant la manière dont un composant logiciel peut communiquer avec un autre.

[2] Object Request Broker, composant principal des intergiciels basés sur les requêtes, chargé de centraliser ces requêtes.

de communication est alors ardu, puisqu'il est nécessaire de reprendre le code source afin de le modifier si celui-ci est disponible. Dans le cas contraire, des "ponts" entre les différents modèles de communication utilisés doivent être développés sur-mesure.

### *2.2 Déploiement des applications*

Comme nous l'avons dit plus haut, une même application peut être déployée dans des contextes différents. Ainsi, une application de conversation de type "messages instantanés" peut être utilisée par une entreprise sur un réseau local, où l'instantanéité est le but recherché, et également sur des terminaux mobiles, où le délai importe moins que la fiabilité de transmission. Actuellement, il est nécessaire d'utiliser deux applications distinctes, vu que les impératifs de communication sont contradictoires. Cependant le cœur de l'application même n'en est pas modifiée, seule la politique de communication l'est.

Un problème se pose également lors de la réutilisation de composants répartis existants. En effet, il n'est pas rare (cela devient même systématique dans des projets importants) pour une nouvelle application de devoir communiquer avec des services ou objets existants. Or le modèle de communication de ces composants peut être varié (e.g. un Web Service, un EJB[1], un objet CORBA[2]) L'application doit alors utiliser un modèle de programmation compatible avec chacun des modèles de communication utilisés (et donc potentiellement plusieurs modèles de programmation pour gérer tous les modèles de communication).

**En mobilité**

Lorsque l'application répartie développée, ou l'un des composants existants avec lesquels elle communique, se trouve être mobile, des changements de contexte peuvent survenir, i.e. des déconnexions fréquentes ou non, des changements de performances ou de fiabilité du lien réseau[3]. Or même si l'application a été conçue pour répondre à un contexte d'exécution, aucun modèle de communication actuel ne peut garantir une transmission optimale des données en toutes circonstances.

[1] Entreprise Java Bean

[2] Common Object Request Broker Architecture

[3] Bien que la notion de contexte dans nos travaux de recherche se cantonnent au lien réseau, d'autres paramètres sont susceptibles de modifier le choix du modèle de communication à utiliser.

### 2.3 Notion de mobilité

Notre proposition exposée dans ce document a été conçue et est vouée à une utilisation en mobilité, i.e. le déploiement et l'utilisation des applications développées à l'aide de notre infrastructure doivent être capables de s'adapter à un environnement mobile.

La mobilité d'une plateforme d'exécution induit un certain nombre de problèmes sur ses communications. Des changements de médium de communication, ou des déconnexions peuvent intervenir, et provoquent alors des ruptures de communication. C'est ce problème de rupture qui est principalement visé dans nos travaux.

Une extension de la notion de mobilité est celle de "nomadicité". Cette notion introduit en complément de la mobilité la capacité de l'application à présenter un fonctionnement adapté à la présence – ou l'absence – de connexion à un réseau, sous la forme de modes "en ligne" et "hors ligne" (voire de modes intermédiaires). Si cette seconde notion n'est pas directement adressée par nos travaux, l'ouverture du système que nous avons conçu est suffisante pour permettre à l'application d'adopter ce genre de comportement (cf chapitre III.4).

## 3 Proposition

### 3.1 Infrastructure logicielle

Pour répondre à cette problématique, nous proposons notre infrastructure logicielle multi-modèles. Celle-ci se présente sous la forme d'un modèle de programmation, et autorise l'application ainsi développée à utiliser plusieurs modèles de communication. Les instructions de répartition sont évaluées lors de l'exécution de l'application, et interprétées en fonction du contexte afin de répondre au mieux aux politiques d'adaptation et de déploiement.

### 3.2 Modèle de programmation

Le modèle de programmation proposé est basé sur les annotations Java, afin de fournir une API volontairement simple à mettre en œuvre, mais présentant un panel riche et varié de mécanismes représentatifs des modèles de programmation existants. L'utilisation des annotations permet de plus de présenter une séparation nette entre le code de répartition et le code métier de l'application. Enfin, cette application peut également être exécutée sans

notre infrastructure multi-modèles, et adopte alors un comportement local, dénué de toute répartition.

### *3.3 Modèles de communication*

Comme il sera développé un peu plus loin, différents modèles de communication existent et sont adaptés à un contexte bien précis de déploiement et d'exécution. Nous proposons dans notre infrastructure une combinaison de modèles de communication, capable d'une part de communiquer avec des composants variés utilisant différents modèles, et capable d'autre part d'établir avec d'autres composants multi-modèles des communications optimales[1] en fonction du contexte.

### *3.4 Approche multi-modèles*

Nous appliquons une approche multi-modèles à la définition et à l'implémentation de notre infrastructure afin de fournir une abstraction du modèle de programmation et de communication utilisé. Ainsi l'infrastructure permet d'une part de découpler le modèle de programmation du modèle de communication utilisés, et d'autre part de découpler le modèle de programmation du client de celui du serveur. Il en résulte une indépendance totale entre la gestion des communications pour le client, celle du serveur, et le choix effectif du ou des modèles de communications utilisés pour acheminer ces communications.

## *4 Organisation du document*

Ce mémoire est organisé comme suit :
Dans un premier temps, la définition d'un intergiciel de communication permettra de situer nos recherches et de centrer notre attention sur ses tenants et aboutissants. Ensuite, un panorama de l'état de l'art recensera les différents travaux et produits disponibles aujourd'hui en ce qui concerne le développement d'applications réparties en général, et proposant des mécanismes d'adaptation en particulier.
Nous présenterons ensuite l'approche multi-modèles comme nous la concevons, puis l'infrastructure logicielle multi-modèles que nous proposons pour concrétiser cette approche. Les travaux d'application et d'implémentation de notre infrastructure seront ensuite éxpliqués, ainsi que les résultats visibles de nos travaux. Une conclusion permettra de tirer un bilan sur ces recherches, et d'ouvrir de nouvelles perspectives.

[1] communications présentant le meilleur compromis rapidité/fiabilité

# I

# État de l'art

La communication a été, et est encore le centre d'intérêt principal s'agissant de la programmation. La communication entre l'ordinateur et l'être humain tout d'abord, avec des systèmes basiques de cartes perforées, puis des IHM[1] de plus en plus élaborées ; et très rapidement ensuite la communication entre les ordinateurs (liaisons sérielles, puis interfaces réseaux dédiées). Avec l'apparition d'architectures capables d'exécutions multi-tâches, la communication au sein d'une même plateforme est devenue également une priorité.

Parallèlement aux besoins croissants de communication, les langages et architectures se sont enrichis et complexifiés, offrant autant de nouvelles solutions que de nouveaux verrous. Aujourd'hui encore, les évolutions logicielles présentent de nouveaux défis à relever pour les systèmes de communication en place. En effet, le service de communication nécessaire à une application élémentaire basée sur des appels procédures statiques ne demande pas les mêmes spécificités qu'un service adapté à une application à base de composants capables de renégocier leurs contrats (cf. section I.3.1, le modèle ACCORD)

**IPC**

Le service de communication pour une application peut prendre plusieurs formes. La plus "locale" de ses formes est le principe de communication interprocessus [Lam86]. Sous ce terme général se cachent toutes les techniques

[1] Interface Homme / Machine

utilisées entre processus afin de se synchroniser ou d'échanger des données.

Ces techniques peuvent être regroupées en deux principales catégories. D'une part, les mécanismes de synchronisation entre processus permettent d'organiser leur exécution : les verrous (ou `mutex`), les sémaphores, ou bien encore les signaux fournissent des possibilités d'utilisation exclusive d'une ressource, ou de déclenchement de fonction.

D'autre part, les mécanismes de communication permettent aux processus d'échanger des données : les fichiers, les mémoires partagées, ou bien encore les tubes créent un espace d'échange, voire de dialogue entre eux.

Dans tous les cas de communication inter-processus, les mécanismes sont très élémentaires et demandent un certain travail de développement pour obtenir une communication construite. De plus, ils deviennent complètement inutilisables dans le contexte d'applications réparties, n'étant exploitables que sur une même plateforme d'exécution. C'est pourquoi nous ne traiterons pas d'avantage de ce type de communications, même s'il était important d'en faire mention pour présenter la globalité des moyens de communication possibles.

**Socket TCP et UDP**

Depuis que les interfaces réseau ont été standardisées autour de la couche IP[1], les *sockets* sont la brique de base utilisée par toutes les applications réparties sur un réseau. Ils consistent en une connexion plus ou moins explicite entre deux applications : l'une est le serveur, offrant une connexion disponible à l'autre, le client, qui s'y connecte en s'adressant à la bonne adresse IP[2] et au bon port.
Les connexions *socket* sont donc des connexions de type client-serveur, obligeant le client à connaitre l'adresse du serveur, et le port accessible.

Les deux principaux modes de communications via une connexion socket sont TCP [Pos80a] (appelé alors TCP/IP) et UDP [Pos80b]. TCP (pour Transmission Control Protocol) est un mode de communication dit "connecté", i.e. le client ouvre une session avec le serveur, puis échange des données

[1] D'autres couches existent néanmoins, e.g. AppleTalk, mais la généralisation d'IP en fait le standard actuel en tant que couche 3 du modèle OSI.

[2] Jusqu'à récemment, les adresses étaient dites IPv4 (codées sur 4 octets) ; on voit désormais de plus en plus de systèmes au standard IPv6 (adresses codées sur 16 octets, donc $2^{8x16}$ adresses possibles)

fiables (avec contrôle d'erreur), et la session prend fin soit de la volonté de l'un ou l'autre, soit au bout d'un temps limite ("time out"). UDP est dit "non connecté" car il n'existe aucun concept de session. Le client envoie des paquets de données au serveur, qui peut les recevoir (ou non) dans l'ordre (ou non). Bien qu'étant moins fiable que TCP, UDP présente cependant le grand intérêt de fournir une rapidité de transmission supérieure. Ce type de communication est donc recommandé pour des applications où la rapidité de transfert importe plus que l'exhaustivité des données (comme du streaming[1] par exemple).

Tous les modèles de communication énoncés par la suite se basent sur des communications TCP/IP (ou UDP). En effet, si certains modèles utilisent la communication interprocessus lorsque celle-ci est possible, le fonctionnement de ces modèles se base sur les possibilités des communications réseau par IP (identification des plateformes d'exécution des applications par leur adresse IP par exemple).

**HTTP**

Si les communications TCP et UDP peuvent être utilisées telles quelles, beaucoup d'applications utilisent un protocole de communication afin d'assurer une compatibilité avec d'autres applications. Sur Internet, le protocole de communication généralement utilisé est HTTP [BLFFN96], pour *HyperText Transfer Protocol* sur TCP/IP. L'intérêt original de ce protocole dans sa première version[2] est de fournir un système de requête basée sur une URI[3]. Ainsi le client se connecte au serveur sur un port standard (le port 80 par défaut), puis demande une ressource par son identifiant ; le serveur envoie en retour la ressource demandée, ou bien un code d'erreur normalisé pour expliquer l'échec de la requête. Bien qu'étant un simple protocole de requête, HTTP permet de transférer les requêtes via différents relais que sont les proxies, les passerelles ou bien encore les tunnels.

Dans sa version principale[4], HTTP introduit des notions essentielles, enrichissant grandement l'intérêt de ce protocole. Tout d'abord, le client peut envoyer des données, en modifier ou en supprimer, en plus de pouvoir en recevoir. De plus, une gestion des types MIME[5] permet au serveur de spécifier

[1] Diffusion d'un flux multimédia
[2] HTTP Version 0.9
[3] Uniform Resource Identifier, RFC 3986
[4] HTTP Version 1.0, RFC 1945
[5] Multipurpose Internet Mail Extension, RFC 2077

le type de données envoyées (e.g. du texte, une image, un contenu compressé, un flux). Le transfert des requêtes est encore plus abouti dans cette version, où le serveur peut indiquer par un code d'erreur particulier que la requête doit être transférée à un autre serveur, ou tout du moins à une autre URI.

Dans sa version la plus récente[1], HTTP affine ces notions en offrant la possibilité de négocier le type de données échangées (le client peut spécifier le type qu'il souhaite recevoir). L'évolution marquante de cette dernière version est surtout la persistance des connexions. En effet, cette version du protocole autorise un client à faire, lors d'une seule et même connexion, plusieurs requêtes. Celles-ci peuvent alors être suivies (lorsque la réponse à la première requête est reçue, le client envoie une nouvelle requête), ou bien en cascade (le client envoie plusieurs requêtes à la suite sans attendre les réponses, celles-ci sont envoyées à la chaine par le serveur).

**SOAP**

Comme HTTP, SOAP [BEK+00] (originellement Simple Object Access Protocol) est un protocole de communication. Cependant, il nécessite un autre protocole plus bas niveau afin d'être mis en œuvre. Ainsi, son utilisation se fait le plus souvent avec HTTP, mais il peut également être associé à SMTP[2] afin d'acheminer les requêtes via un serveur de messagerie electronique.
SOAP est un protocole basé sur le langage balisé XML[3]. Les messages SOAP sont composés de deux parties :

- Une enveloppe, contenant des informations sur le message lui-même afin de permettre son acheminement et son traitement,
- un modèle de données, définissant le format du message, i.e. les informations à transmettre

Le principal intérêt de SOAP est de permettre un bonne interopérabilité entre applications réparties de type et d'origine différents. En effet, le langage XML permet d'abstraire complètement la plateforme d'exécution ou le langage de programmation du client ou du serveur, puisqu'il s'agit d'un langage autodescriptif. Cependant l'utilisation du XML apporte aussi ses limitations ; chaque message étant richement rempli d'informations, de descriptions et de méta-données, le tout en texte humainement lisible, la taille (et donc le temps de transfert) de ce message est rapidement conséquente.

[1] HTTP Version 1.1, RFC 2616
[2] Simple Mail Transfer Protocol, RFC 2821
[3] eXtensible Markup Language

La communication inter-processus, et les modèles de communication "bas niveau" que nous venons de présenter sont la base des communications entre applications. Les développeurs d'applications doivent, pour les utiliser, concevoir des systèmes clients et serveurs implémentant ces mécanismes. Ceux-ci présentent une interface d'utilisation relativement sommaire, et demandent donc un temps de développement conséquent. Ce temps consacré à l'utilisation des communications peut être considéré comme "non productif" pour le code métier. C'est pourquoi, comme nous allons le voir, l'utilisation des intergiciels, qui réduisent ce temps de développement "non productif" par l'abstraction des communications dans l'application, permet au développeur de se concentrer sur le code métier.

## *I.1 Intergiciel de communication : définition*

Le terme *middleware* semble avoir fait sa réelle apparition vers les années 1990 [Kra06], bien qu'il semble exister depuis 1968 [NR68]. Le terme *intergiciel* apparut une décennie plus tard[1]. Le principe de l'intergiciel a donc fait son apparition il y a bien longtemps : la nécessité de simplifier les développements en offrant une abstraction des spécificités de chaque plateforme d'exécution a toujours été présente, et de nombreux outils permettant cette abstraction ont vu le jour sans officiellement s'être vu donné le nom d'intergiciel.

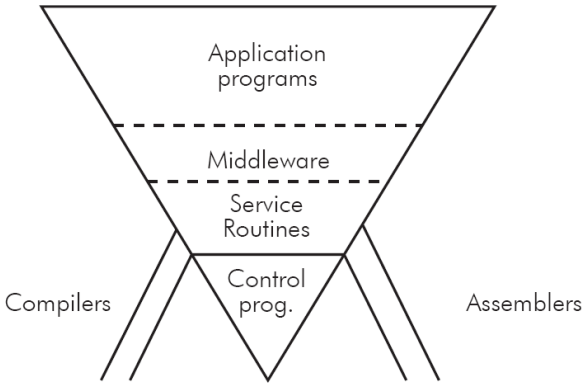


*Fig. I.1:* Pyramide inversée de d'Agapeyeff [NR68].

[1] Officiellement, le terme adéquat pour désigner ce type d'outil de communication est "logiciel médiateur", d'après la Délégation Générale à la Langue Française et aux Langues de France.

Un intergiciel est donc un programme situé entre l'application et les services bas niveau (cf figure I.1).

### *I.1.1 Description d'un intergiciel [de communication]*

Un intergiciel se présente généralement sous la forme de deux composants principaux : un modèle de programmation (également appelé personnalité applicative, cf Préface) et un modèle de communication (ou personnalité protocolaire). Jusqu'à aujourd'hui, ces deux modèles sont fortement couplés, i.e. le type de communications utilisées par l'intergiciel est reflété par son modèle de programmation. Un intergiciel présente par le biais de son modèle de programmation un certain nombre de services, certains obligatoires, d'autres facultatifs.

#### *Modèle de programmation*

Le modèle de programmation d'un intergiciel définit comment le développeur peut utiliser l'intergiciel pour assurer les communications entre composants logiciels. Il peut offrir une API[1] et/ou des mécanismes découplés de la programmation traditionnelle, e.g. une description externe d'interfaces comme IDL (Interface Definition Language), ou annotations comme SCA (Service Component Architecture, cf section I.5.3).

Ce modèle de programmation peut être profondément intégré dans le langage, permettant d'assurer implicitement des communications sans manipuler de concepts étrangers : un simple héritage de classe permet dans certains cas de rendre l'objet accessible aux autres. Le modèle peut également rester basique, permettant d'assurer les communications via des méthodes dédiées, mais nécessitant la gestion explicite de ces communications par les objets. L'intérêt de chaque type de modèle est justifié : certains préfèreront s'abstraire complètement des mécanismes de communication, alors que d'autres préfèreront garder le contrôle sur ceux-ci.

#### *Modèle de communication*

L'intergiciel permet au développeur de gérer les communications implicitement ou explicitement au niveau de l'application grâce à son modèle de programmation. Il doit de plus assurer la communication entre différentes applications via son modèle de communication.

[1] *Application Programming Interface*, interface de programmation d'application

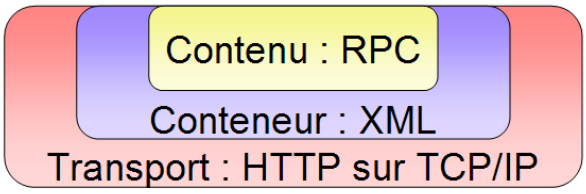


*Fig. I.2:* Différentes couches d'un modèle de communication : exemple du XML-RPC.

Comme symbolisé en figure I.2, ce modèle se compose d'un moyen de transport, d'un conteneur et d'un contenu :

- Le contenu est généralement très proche du modèle de programmation associé : il s'agit de la communication applicative proprement dite.
- Le conteneur permet au modèle de communication de formater le contenu. Il peut s'agir d'une syntaxe ou d'un codage particulier.
- Le moyen de transport est la couche physique (ou proche de la couche physique) sur laquelle la communication se repose. Il s'agit généralement d'une communication TCP/IP ou d'une surcouche de cette communication (e.g. HTTP, SMTP).

*Services proposés*

Le modèle de programmation est obligé d'offrir certains services afin d'assurer des communications efficaces. Ainsi les services de communication, de nommage et de liaison (*binding*) [Qui03] sont indispensables à l'intergiciel pour référencer et communiquer.

Des services annexes sont fournis quelquefois par les intergiciels, permettant de contrôler plus précisément le cours de l'exécution et les échanges de données. Parmi ces services, on trouve des outils presque indispensables comme le service de persistence [Kra08], ou bien encore la sécurité, la qualité de service ou les transactions.

## *I.2 Intergiciels "mono-modèle"*

Des intergiciels de communication existent pour réduire – voire supprimer – le temps de développement relatif à la gestion des communications dans une application. Nous allons recenser ici tout d'abord les principaux intergiciels de communication ne présentant qu'un unique modèle de programmation /

communication (ce qui représente la grande majorité des intergiciels utilisés aujourd'hui).

### *I.2.1 Web Services*

L'intergiciel de communication le plus "en vogue" de nos jours est sans nul doute le Web-Service [CKM+03]. Il s'agit plus exactement d'une spécification, mais de nombreux intergiciels l'implémentent afin de permettre simplement et efficacement de fournir et d'utiliser un Web-Service. Parmi ceux-ci, JAX-WS[1], Weblogic, Websphere, ou bien encore le framework .Net (et donc tous les langages de programmation .Net comme C#, J#).

Le Web-Service est basé sur un modèle de programmation de type RPC (Remote Procedure Call, appel de procédure à distance). Dans un premier temps, le client cherche les Web-Services disponibles (grâce à des méthodes de découverte), puis il consulte la description du service voulu, et enfin il envoie une requête à ce service, qui lui envoie en retour la réponse à sa requête. La particularité de ce modèle de programmation est que les Web-Services sont sans état ("stateless"), i.e. la requête est indépendante de toute session (tout du moins du point de vue du Web-Service), et le cycle de vie du service équivaut à la durée de la requête.

Le modèle de communication utilisé se base sur UDDI[2] pour la découverte de services, le langage WSDL[3] pour leur description, et un protocole de requête qui peut être SOAP ou XML-RPC[4]. Comme le Web-Service utilise SOAP (et XML-RPC, un "SOAP-like"), le modèle de communication sous-jacent peut varier : (e.g. HTTP, JMS), mais il s'agit presque toujours du protocole HTTP.

### *I.2.2 CORBA*

Si les Web Services sont les plus populaires pour établir des communications entre applications indépendantes (ou tout du moins faiblement couplées), CORBA [GGM99] est l'intergiciel de référence en ce qui concerne les communications de type RPC. En effet, cette spécification officialisée[5] en

[1] Java API for XML-Based Web Services, JSR 224
[2] Universal Description Discovery and Integration
[3] Web Service Description Language
[4] XML Remote Procedure Call
[5] CORBA comme il est utilisé aujourd'hui correspond aux fondements introduits à sa version 2.0

1995 par l'OMG introduit des concepts novateurs et performants, capables d'intégrer les communications au sein même du développement des applications, et ce dans plusieurs langages de programmation.

Le modèle de programmation proposé est basé sur une description d'interfaces neutre (i.e. non dépendante d'un langage de programmation). Le standard IDL a été choisi pour décrire les interfaces ; des mécanismes sont ensuite chargés d'automatiquement générer le code nécessaire à la bonne gestion de ces interfaces.
L'application cliente utilise alors une souche générée pour accéder à des objets distants. Les invocations ne diffèrent pas des invocations classiques, un simple appel de méthode suffit.
L'application serveur implémente le squelette généré pour mettre à disposition un objet distant respectant l'interface de base. L'objet implémentant reçoit alors les invocations également comme des appels de méthode.

Le modèle de communication de CORBA utilise le protocole GIOP[1], et plus particulièrement une instanciation de ce protocole sur TCP/IP, IIOP[2]. L'intérêt principal de ce protocole est qu'il présente une représentation commune des données (CDR[3]) résolvant les différences entre les architectures matérielles natives, ce qui permet une meilleure interopérabilité entre plateformes de différents types (e.g. processeurs à architecture 8, 16, 32, 64 bits, représentation en little-endian / big-endian). IIOP est traditionnellement utilisé sur TCP/IP, profitant des propriétés de sécurité et de contrôle d'erreur de ce dernier.

CORBA est essentiellement connu et utilisé pour son système de RPC fiable, performant et interopérable. La spécification a pourtant au fur et à mesure présenté de nouveaux services et paradigmes, comme la gestion de la qualité de service, l'invocation de méthode asynchrone[4], ou bien encore la communication par évènements.
Si la spécification CORBA est complète et puissante, rares sont les implémentations de CORBA qui intègrent l'entière spécification. Depuis 2003[5], le standard CORBA s'est stabilisé, et un modèle de composant (CORBA Component Model) a également été spécifié afin de permettre la prise en compte

[1] General Inter-ORB Protocol
[2] Internet Inter-ORB Protocol
[3] Common Data Representation
[4] AMI, Asynchronous Method Invocations
[5] Spécification de CORBA v3.0

de concepts plus évolués.

### *I.2.3 RMI*

De nombreux intergiciels proposent, comme CORBA, un modèle de programmation efficace pour gérer les communications dans une application répartie. C'est le cas de RMI [Mic03] (pour Remote Method Invocation), un intergiciel de communication pour le langage Java. La marche à suivre pour développer des objets répartis est un peu différente de CORBA. Comme RMI n'est proposé que pour un seul langage, les descriptions d'interfaces sont directement codées dans celui-ci, ce qui évite de devoir manipuler un second langage de description en plus de la programmation. Pour qu'un objet puisse être accessible à distance, un simple héritage de classes RMI spécifiques suffit. Le grand avantage de RMI est qu'il est inclus dans les bibliothèques standards Java, ce qui permet à toute application Java de pouvoir utiliser RMI sans aucun composant additionnel.

Jusqu'à la version 5.0 de Java, l'utilisation d'un outil nommé *rmic*[1] était obligatoire afin de générer les souches et squelettes nécessaires à l'utilisation répartie de l'application. Depuis cette version, un mécanisme d'invocation dynamique permet de passer cette étape.
Le modèle de communication utilisé par RMI est le JRMP[2], un protocole utilisé sur TCP/IP et présentant l'avantage d'être très léger. Cependant, pour rendre RMI compatible avec d'autres intergiciels (notamment CORBA), le développeur peut spécifier à RMI d'utiliser le protocole IIOP pour établir ses communications. Certaines implémentations de RMI utilisent même des protocoles propriétaires (e.g. le protocole ORMI pour Orion, T3 pour WebLogic).

### *I.2.4 .Net Remoting*

Pour les applications développées en .Net, Microsoft fournit un intergiciel de communication sur-mesure : .Net Remoting [MWN02]. Il s'agit d'un modèle de programmation de type RPC allié à un modèle de communication modulable. Comme RMI pour Java, .Net Remoting est inclus dans les bibliothèques standards de .Net, et s'intègre complètement dans le développement de l'application. Une fois les objets instanciés, un appel distant ne se différencie pas d'un appel local classique. Les langages de programmation .Net intègrent des solutions d'invocation de méthode asynchrone. Ces mêmes

[1] Java Remote Method Invocation Compiler
[2] Java Remote Method Protocol

solutions sont également utilisables dans une application répartie, autorisant une réception asynchrone de la réponse par scrutation ou par *callback*[1].

Le modèle de communication de cet intergiciel peut être de différents types, selon le choix du développeur. Comme .Net Remoting ne concerne que les langages et plateformes .Net, la question d'interopérabilité ne se pose pas dans la plupart des cas d'utilisation. C'est pourquoi le protocole de communication par défaut est un protocole propriétaire. L'utilisation de SOAP comme protocole est toutefois possible. Ces protocoles peuvent être employés soit sur une connexion TCP directe, soit sur une connexion HTTP permettant notamment leur utilisation malgré la présence de pare-feu ou autres proxies. Il est intéressant de noter que l'utilisation de SOAP sur HTTP permet d'assurer la compatibilité avec les Web Services, mais l'absence d'utilisation de IIOP empêche toute interaction avec des intergiciels du type CORBA/RMI.

### *I.2.5 JMS*

Si les intergiciels de type RPC sont légion, ceux proposant d'autres paradigmes de programmation et de communication sont nettement plus rares. JMS [Mic02] (pour Java Message Service) est la spécification d'un service de message pour le langage Java. Il s'agit en fait essentiellement d'une spécification de modèle de programmation, autorisant une application développée avec JMS de pouvoir utiliser indépendamment n'importe quelle implémentation de cette spécification. Le ou les modèles de communication utilisés sont par contre complètement libres selon l'implémentation choisie, ce qui pose évidemment un problème de compatibilité lors de l'utilisation simultanée d'implémentations différentes.

Le modèle de programmation JMS définit deux types de communications asynchrones. Dans un premier temps, les communications point à point de type "file de messages" permettent une liaison fiable entre composants répartis (i.e. les messages sont stockés sur le serveur tant qu'un consommateur ne les en aura pas retirés). Pour le consommateur, plusieurs moyens sont fournis afin de gérer les messages :

- Le consommateur peut scruter la file de messages pour savoir si elle contient des données,
- Il peut alors "consommer" le premier message.

[1] Méthode choisie lors de l'invocation, qui sera invoquée en retour lorsque la réponse sera reçue.

– Il peut également fournir un objet de type *MessageListener* afin de gérer automatiquement les messages lorsqu'ils sont reçus.

Dans un second temps, les communications de type "publish and subscribe"[1] permettent à un ou plusieurs "publicateur(s)" d'envoyer des messages à un ou plusieurs abonné(s). Une fois abonné, le composant "consomme" les messages comme pour une file de messages.

Cet intergiciel, comme la grande majorité des intergiciels proposant des modèles asynchrones, se base sur l'utilisation d'un (ou de plusieurs) serveur intermédiaire, chargé de stocker et d'acheminer les messages. Ce système présente des bons et mauvais côtés : d'une part il permet aux applications de s'abstraire du stockage, de la gestion des destinataires et autres tâches inhérentes au serveur, mais d'autre part il alourdit le processus de communication en ajoutant un intermédiaire entre les composants répartis (ce qui multiplie le nombre de communications sur le réseau).

### *I.2.6 JavaSpaces*

Autre intergiciel de communication asynchrone, JavaSpaces [FHA99] propose un système d'espaces partagés afin de permettre aux applications réparties de travailler sur des données communes. La force de cet intergiciel, en plus de proposer un type de communication asynchrone, réside dans le fonctionnement de ces espaces partagés. Tout d'abord, plusieurs composants peuvent accéder à un même espace partagé (c'est d'ailleurs l'intérêt premier du principe). De plus, cet espace est persistant ; les objets insérés y restent indéfiniment, ou bien pendant un certain temps (spécifié par le propriétaire de l'objet).

La technique utilisée pour trouver un objet dans l'espace partagé en est le point le plus intéressant. En effet, la recherche est dite "associative" car elle permet d'une part de demander un certain type de données (i.e. un nom de classe d'objet), mais également d'exiger certaines valeurs (i.e. des attributs de l'objet doivent correspondre à ceux de la requête). Cet espace partagé peut être ainsi comparé à une base de données sur laquelle des requêtes peuvent être formulées.

La cohérence des données est assurée par un principe simple : les objets contenus dans un espace ne peuvent être modifiés, mais peuvent être retirés pour être modifiés localement, puis réinsérés dans l'espace ; ceci évite tout problème transactionnel de modifications simultanées.

[1] littéralement "publie et souscrit"

Tout comme JMS, cet intergiciel utilise un ou plusieurs serveurs intermédiaires. Ce serveur gère les espaces partagés en stockant leur contenu, et en assurant les mécanismes d'ajout, de suppression et de consultation par les composants.

### *I.2.7 Synthèse des intergiciels mono-modèles*

Des intergiciels présentant un unique modèle de programmation et de communication existent en nombre important, et assurent des communications fiables et efficaces entre composants répartis dans leur contexte de prédilection. Ils permettent d'utiliser des mécanismes synchrones et asynchrones pour donner la possibilité aux applications de communiquer entre elles, en abstrayant la couche réelle de communication du système.

Si leur utilisation est suffisante pour le développement et le déploiement de la plupart des applications réparties, leur aspect "figé" ne permet pas de modifier leur comportement pour s'adapter au contexte d'exécution.

## *I.3 Intergiciels réflexifs / multi-modèles*

Depuis plusieurs années maintenant des équipes de recherche (ainsi que des industriels) s'intéressent aux limitations des intergiciels conventionnels, et donc également à dépasser ces limites en explorant de nouveaux principes. Parmi ces nouvelles idées, deux tendances se font sentir :

- Les intergiciels réflexifs, ou comment un intergiciel peut s'observer lui-même pour modifier son comportement,
- Les intergiciels multi-modèles, ou comment tirer le meilleur parti des intergiciels existants en les combinant.

Dans ces deux cas, la raison des modifications (ou combinaisons) apportées est le changement de contexte. Avant de recenser ces principaux intergiciels "nouvelle génération", réflexifs et multi-modèles, nous allons citer les principales solutions existantes pour permettre à une application de s'adapter au contexte.

### *I.3.1 Adaptation au contexte*

Afin de fournir un moyen de communication fiable et performant en toutes circonstances, de nombreux travaux ont été menés afin de permettre aux programmes en général, et aux intergiciels en particulier, de s'adapter au contexte. Pour cela, différents moyens existent aujourd'hui [Sar05], donnant

aux applications la capacité d'adapter leur comportement en fonction des possibilités de leur environnement.

**L'adaptation au niveau composant**
Dans des applications à base de composants, certains modèles (comme le modèle ACCORD [BCTT05]) proposent des contrats[1] dits *pragmatiques*, i.e. basés sur l'environnement dans lequel les composant est utilisé. Lorsque cet environnement évolue, les contrats sont renégociés, ceci pouvant aboutir sur un assemblage différent. L'application s'est alors adaptée au contexte.

**Programmation orientée aspect (AOP)**
Dans une optique de séparation des préoccupations (originalement *Separation of Concerns, SoC*), la programmation par aspects [KLM+97] permet de séparer des préoccupations transverses (originalement *Cross-cutting concerns*, e.g. la sécurité, la communication) afin de clarifier le développement et la maintenance d'un programme. Une fois les aspects définis et codés, l'étape de tissage permet d'obtenir le programme final.
Lors d'un changement de contexte, les préoccupations affectées par cette évolution sont identifiés comme aspects. Deux types d'adaptation sont alors possibles :

- A l'étape du développement / de la compilation du programme ; seul l'aspect affecté par le changement doit être modifié, le programme est ensuite tissé puis exécuté. plusieurs outils orientés aspects permettent ceci, comme Spoon [Paw06].
- A l'étape de l'exécution du programme ; JAC [PDF+02] est un outil de programmation par aspects pour Java, alliant la réflexion et les aspects, afin de permettre aux applications de modifier leur comportement par préoccupation (e.g. si un arrêt prochain de la plateforme est envisagé, l'aspect de persistance est activé pour sauvegarder les données).

**Protocoles basés sur des méta-objets : réflexion**
Le moyen le plus utilisé aujourd'hui pour l'adaptation au contexte, en particulier en ce qui concerne les intergiciels, est la réflexion : l'intergiciel, par réflexion, est capable de réifier des concepts ou notions abstraites, et ainsi d'y accéder via des objets. On parle alors de méta-objet [KDRB91]. De nombreux langages de programmation intègrent aujourd'hui la réflexion, comme SmallTalk [Riv96] ou bien encore Java (à un niveau moindre, cependant). Ces langages permettent d'introspecter le programme via ses classes (dans

[1] ACCORD est un modèle de composants qui propose un assemblage par contrats

les langages objets), voire d'intercéder[1] dans certains cas, ce qui permet de grandes facultés d'adaptation.

### *I.3.2 SCA : l'OpenSOA*

> Implementing SOA for the first time is the triumph of imagination over intelligence. Implementing SOA for the second time is the triumph of hope over experience.
>
> SOA Facts.com

Le SOA (pour *Service Oriented Architecture*) est un modèle d'interactions entre services. Il définit un lot de règles que les services et l'architecture doivent respecter. On trouve de nombreuses propriétés, dont une grande partie définissaient déjà les Web Services, comme le couplage faible entre les services, ou encore l'unicité de l'instance d'un service. De ce modèle d'interactions, le consortium OpenSOA [Cha07] a définit une spécification appelée SCA (pour *Service Component Architecture*).
Si le SCA n'est qu'une spécification, plusieurs implémentations et projets concrets existent, fournis par les principaux partenaires du consortium et d'autres organismes privés et publics. On peut citer entre autres Tuscany [Tus08] et FraSCAti, une plateforme d'exécution développée dans cadre du projet SCOrWare [AWE+08].
Le principal intérêt de SCA par rapport à une architecture de Web Services "classique" est l'utilisation d'un bus de communication (plus précisément un ESB, pour *Entreprise Service Bus*), permettant l'utilisation implicite de plusieurs modèles de communication. De plus, la spécification est assez ouverte et complète pour permettre des implémentations en une multitude de langages de programmation, tels que Java, Spring, BPEL, C++, Cobol, ainsi qu'une multitude de modèles de communication comme les Web Services, JMS, EJB, JCA.
On se rend clairement compte du potentiel offert par SCA : une indépendance de plateforme et de langage comme CORBA, mais la multiplicité des modèles de communication en plus, ce qui permet d'utiliser des services utilisant différents modèles. Cependant, aucune proposition n'est faite pour combiner ces modèles, ou d'adapter leur choix en fonction des changements de contexte.

[1] Si l'introspection permet d'observer l'état et le comportement du programme, l'intercession permet d'en modifier le comportement

### *I.3.3 FlexiNet*

FlexiNET [HHD98] est un intergiciel indépendant, synchrone, pouvant dialoguer d'une part avec des applications de type CORBA (via le protocole IIOP), et d'autre part avec des applications REX [OO87], un autre modèle synchrone d'invocation de procédures à distance basé sur UDP

Ce modèle basé sur Java se présente sous la forme d'une suite de filtres, au travers desquels les messages reçus et émis doivent transiter. Ces filtres, à la manière des aspects pour les interceptions d'appels de méthodes, sont donc consultés un à un lorsqu'une invocation à distance se produit. Chacun de ces filtres a alors la possibilité de contrôler et de modifier les paramètres, la source et la destination de l'invocation. Ce modèle étant réflexif, chaque filtre est en mesure de se modifier lui-même, d'ajouter un filtre supplémentaire ou d'en retirer. Pour que les filtres puissent intercepter les invocations simplement, ces dernières leur sont transmises sous la forme de messages, intégrant les différentes données nécessaires au traitement de l'invocation telles que la source, la destination et les paramètres.

L'intérêt supplémentaire de ce modèle réside dans les deux modèles de communication qu'il peut adopter. Ces deux modèles peuvent être utilisés tour à tour, au gré des sollicitations par les filtres présents. Dynamiquement, un filtre peut décider d'utiliser soit le modèle REX, soit le modèle CORBA, que ce soit en émission ou en réception d'invocation.

### *I.3.4 PolyORB*

Sur un principe équivalent à FlexiNET, PolyORB [VHPK04] est un intergiciel hybride compatible avec plusieurs modèles de communication. Ce dernier propose de choisir le modèle de communication à utiliser lors du lancement de l'application, et non lors de sa conception. Le modèle de programmation associé permet donc à une application développée en Ada95 d'établir des communications synchrones comme asynchrones. Cet intergiciel intègre plusieurs personnalités, comme CORBA, SOAP, ou bien encore un modèle orienté message appelé MOMA, basé sur JMS. Partant de cet aspect multi-personnalités, l'appellation d'"intergiciel schizophrène" utilisée par ses concepteurs résume assez bien le principe. En effet PolyORB, basé sur un seul et même modèle de programmation, peut prendre plusieurs personnalités selon le contexte d'exécution.

Le point principal de cet intergiciel est son modèle de programmation, qui intègre toutes les composantes nécessaires à un modèle de communication complet, comme le référencement, l'activation, l'aiguillage, tout en restant suffisamment neutre pour pouvoir utiliser un modèle de communication synchrone ou asynchrone. Ce dernier peut être choisi au lancement de l'application, cependant PolyORB ne permet pas d'adopter un comportement réflexif au cours de l'exécution, l'empêchant ainsi de modifier ce choix une fois l'application lancée.

### *I.3.5 DynamicTAO*

TAO[1] est une implémentation d'une grande partie de la spécification CORBA, incluant *Real-time CORBA*, qui fournit une qualité de service efficace et prédictible, adaptée au déploiement sur des systèmes temps-réel.
DynamicTAO [KRL+00] est une évolution de TAO permettant à l'ORB de se reconfigurer dynamiquement par un comportement réflexif. Si ses possibilités sont grandes, les fonctions essentiellement exploitées dans cet intergiciel sont celles d'inspection et de configuration à distance via l'interface ad hoc implémentant le protocole DCP[2]. Le type d'adaptation et de reconfiguration permises par DynamicTAO sont ainsi distantes et centralisées, permettant de superviser interactivement le déploiement. Mais les mécanismes autorisant une réelle adaptation autonome au contexte sont à l'heure actuelle non implémentés.

Le développement de DynamicTAO est arrêté depuis quelques années, une évolution du principe a pris sa suite sous la forme de LegORB, un ORB basé sur des bibliothèques dynamiques, offrant un intergiciel compatible avec le modèle de communication de CORBA (IIOP) dont les composants sont connectables et déconnectables à l'exécution pour assurer sa configuration dynamique (i.e. l'activation / désactivation de ses services).

### *I.3.6 Jonathan*

Les intergiciels comme DynamicTAO démontrent l'intérêt d'un comportement réflexif pour permettre de se reconfigurer et de s'adapter aux changements en cours d'exécution. Une autre approche du problème d'adaptation proposée par Jonathan [DHDTS98] est le développement d'un noyau d'intergiciel générique. En effet, si la réflexion permet à un intergiciel de modifier

[1] The ACE (ADAPTIVE Communication Environment) ORB
[2] Distributed Configuration Protocol

son comportement, certains cas nécessitent purement et simplement l'utilisation d'un autre modèle de communication pour s'adapter. C'est ce que propose Jonathan, en fournissant une base commune au développement d'intergiciels, incluant les mécanismes définissant l'ORB, comme le *binding* et la communication.

Pour justifier le développement de Jonathan et ainsi prouver le bien fondé de leur approche, ObjectWeb propose deux personnalités développées à partir de Jonathan :

- David est une personnalité implémentant l'ORB de CORBA. Seulement quelques fonctionnalités des spécifications CORBA sont implémentées, mais le but est ici de prouver qu'une personnalité précise peut être développée autour du noyau de Jonathan.
- Jeremie est une personnalité présentant un style de programmation "à la RMI"

Pour le développement de ces deux personnalités, le noyau Jonathan a représenté la moitié du code écrit (i.e. il n'a été nécessaire de coder que la moitié de l'intergiciel, grâce à l'utilisation de ce noyau).

### *I.3.7 OpenORB et ReMMoC*

Geoff Coulson et al. proposent une plateforme de développement comparable[1] à Jonathan, nommée OpenORB [BCA+01]. Il s'agit d'une architecture de base au développement d'intergiciels, à base de composants, et réflexive. Construite à l'aide d'une partie du modèle de composant COM (la technologie ainsi développée est un modèle réflexif léger de composants nommé Open COM), cette architecture permet l'implémentation d'intergiciels configurables et re-configurables. L'implémentation d'un intergiciel compatible CORBA à l'aide de l'architecture Open ORB a d'ailleurs été développée pour valider cette approche.

Conjointement et à l'aide du développement de cette base réflexive à composants, leurs recherches se sont concentrées sur le développement en collaboration avec Bell Labs UK d'un intergiciel réflexif orienté services appelé ReMMoC [GBS03]. Le modèle de programmation proposé est basé sur les Web Services, et l'intergiciel est capable d'une part de découvrir des services par le biais du protocole UPnP, ou du protocole SLP, et d'autre part

[1] L'intention d'OpenORB et de Jonathan est de fournir un canevas de développement d'intergiciel, en ce sens leur approche est comparable.

d'utiliser ces services (i.e. d'établir la communication) avec des modèles de communication différents comme CORBA, SOAP, ou un modèle de type *publish/subscribe* basé sur SOAP.

### *I.3.8 Commutation synchrone/asynchrone*

Victor Budau et Guy Bernard [BB02] exposent un modèle de communication hybride pour composants EJB[1], basé d'une part sur le modèle synchrone JavaRMI, et d'autre part sur le modèle asynchrone JMS. Ce modèle hybride consiste en un commutateur Synchrone/Asynchrone, interceptant les émissions et réceptions d'invocations, et choisissant le modèle de communication le plus approprié pour transmettre ces invocations. Lorsque l'application veut transmettre une invocation, le commutateur tente l'utilisation du modèle synchrone, et si celui-ci ne réussit pas, le commutateur utilise donc le modèle asynchrone. Inversement, lorsqu'un message doit être transmis par le modèle asynchrone, le commutateur tente d'abord de l'envoyer via le modèle synchrone, pour éviter le surcoût du modèle asynchrone. Si le principe de commutateur est intéressant, les mécanismes encore basiques de commutation basés sur les exceptions méritent un approfondissement. Ce modèle hybride prometteur reste pertinent et justifie l'utilisation de plusieurs modèles de communication selon les circonstances.

### *I.3.9 Synthèse*

Nous venons de présenter un certain nombre de propositions visant à combler le manque de faculté d'adaptation des intergiciels actuels. Les approches sont variées : certains, comme SCA, donnent la possibilité de communiquer avec des services variés. D'autres préfèrent munir l'intergiciel d'un comportement réflexif pour le rendre capable de reconfiguration. Enfin, la notion de personnalités permet de mettre en évidence une base commune aux différents intergiciels, à partir de laquelle les services spécifiques peuvent se greffer.

Si l'intérêt d'utiliser plusieurs modèles de communication afin de communiquer avec plusieurs composants est unanime, aucun n'a relevé l'utilité de combiner plusieurs modèles pour une même communication avec un composant, or cette utilisation peut être un atout certain dans un contexte évoluant sans cesse, comme la mobilité.

[1] Entreprise Java Beans

## *I.4 Principaux modèles de programmation existants*

Comme nous l'avons détaillé dans le chapitre I.1, les intergiciels sont définis d'une part par le ou les modèles de communications qu'ils utilisent, mais également par le modèle de programmation qu'ils fournissent. De ce modèle dépendra la facilité d'utilisation et donc l'acceptation par les développeurs de l'intergiciel plutôt qu'un autre.

Les projets cités au chapitre I.3 donnent une grande importance au modèle de communication utilisé, et visent à permettre de communiquer avec tous les modèles possibles. Toutefois, le modèle de programmation le plus souvent préféré est le modèle RPC, mieux intégré au développement classique de la programmation objet. Ce choix n'est pas forcément judicieux, car chaque modèle de programmation possède ses spécificités qui en font le modèle adapté à un type de programmation, et utiliser un modèle RPC en toutes circonstances peut rendre l'exécution des programmes moins performante dans certains cas.

***Modèles synchrones***

### *I.4.1 Les signaux*

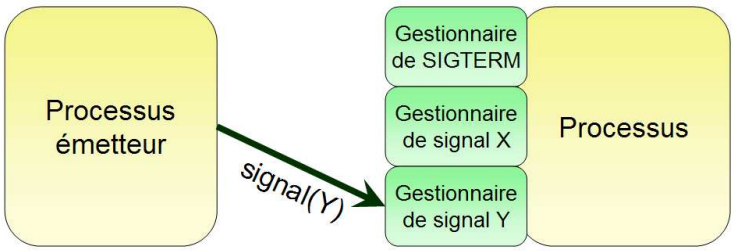


*Fig. I.3:* Principe de communication par signaux.

Les signaux sont un moyen direct et immédiat pour communiquer entre processus ou plus globalement entre programmes exécutés sur la même plateforme. Selon la plateforme (et le système d'exploitation) sur laquelle les programmes sont exécutés, la manière d'envoyer un signal ainsi que son type peuvent varier, mais le principe reste le même :
Un processus appelant envoie un signal à un autre processus en précisant son identifiant. Le processus appelé qui a précédemment spécifié une méthode

*callback* (désigné par "Gestionnaire de signal" sur le schéma) correspondant au signal voit cette méthode appelée, avec comme paramètre le type de signal.

Ce type de communication reste assez sommaire, et, surtout, ne concerne que les processus exécutés sur la même plateforme. C'est pourquoi, même si leur présentation est nécessaire, nous n'allons pas les traiter dans la suite de nos travaux.

### *I.4.2 Modèle RPC*

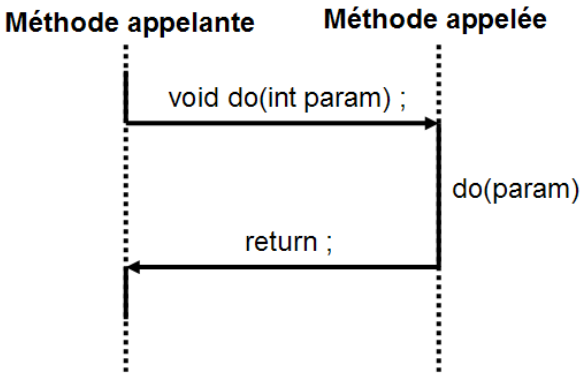


*Fig. I.4:* Principe de communication synchrone RPC.

Comme il a été dit plus tôt, le modèle de programmation RPC tend à devenir le modèle de référence des intergiciels "nouvelle génération" comme il l'est également actuellement (les Web Services, les EJB, RMI et CORBA étant les intergiciels principalement utilisés aujourd'hui). Son principe peut être résumé par un système de requête / réponse synchrones. Si ce modèle est populaire et bien accepté des développeurs, c'est en grande partie grâce à son importante similarité avec le modèle de programmation des langages procéduraux : lorsqu'on appelle une méthode, on lui soumet des paramètres (on obtient alors une requête) ; on reçoit en retour une valeur (la réponse).

Le modèle de programmation RPC fourni par les intergiciels peut être de deux types : l'appel à l'intergiciel pour gérer l'appel de méthode peut être implicite ou explicite.
Dans le premier cas, les objets (ou directement les méthodes pour les langages non-objet) répartis et locaux ne présentent aucune différence[1] pour le

[1] Aucune différence n'est notable une fois l'objet réparti instancié

développeur. L'appel de méthode est effectué exactement de la même façon sur les deux types d'objets. L'utilisation de la valeur de retour l'est également. Cet appel implicite permet au développeur de garder ses habitudes de programmation sans se soucier de la répartition, ce qui optimise et facilite son développement.
Dans le deuxième cas, l'appel de méthode distante est effectué explicitement auprès d'un gestionnaire de communication (généralement l'ORB). On fournit à ce gestionnaire la référence de l'objet distant (e.g. URI, corbaloc), le nom de la méthode à invoquer ainsi que les paramètres. Si ce type de programmation est plus complexe à mettre en œuvre, il présente l'intérêt de connaître réellement la répartition de l'application, et de contrôler point par point les communications entre les composants. Néanmoins, ce modèle explicite est rarement utilisé car non justifié dans la plupart des cas.

Afin de permettre l'invocation directe d'objets répartis, le modèle de programmation RPC doit fournir des objets locaux et distants identiques au programmeur. Ceci est fait par un mécanisme de génération automatique de classes : une fois l'interface de l'objet réparti définie, une étape intermédiaire génère des classes "proxy", permettant au gestionnaire de communication de relayer les invocations. Ainsi l'objet proxy correspondant à l'objet distant reçoit les appels de méthodes, et les transmet via le gestionnaire à l'objet distant. Par héritage, les objets proxy peuvent être manipulés similairement aux objets originaux.
Présentant le deuxième modèle de programmation RPC explicite, des interfaces d'invocation dynamiques (e.g. la DII de CORBA, Dynamic Invocation Interface) permettent d'invoquer des méthodes distantes sans être passé par l'étape de génération de classe proxy. L'intérêt de cette technique est de pouvoir communiquer avec des objets que l'on ne connaissait pas lors de la compilation de l'application. Ceci présente un atout certain dans des systèmes de découverte dynamique de services.

#### *Modèles asynchrones*

### *I.4.3 Les files de messages, "producteur/consommateur"*

Le modèle de programmation basé sur les files de messages présente des objets dédiés. Ainsi il existe deux moyens d'instancier une file de messages ; soit en demandant sa création, soit en soumettant la référence (i.e. le nom) d'une file existante au système. Une fois cette file instanciée, le composant se déclare soit producteur, soit consommateur, soit les deux. Des règles d'identification régissent cette déclaration, qui peut ainsi être refusée en cas d'échec

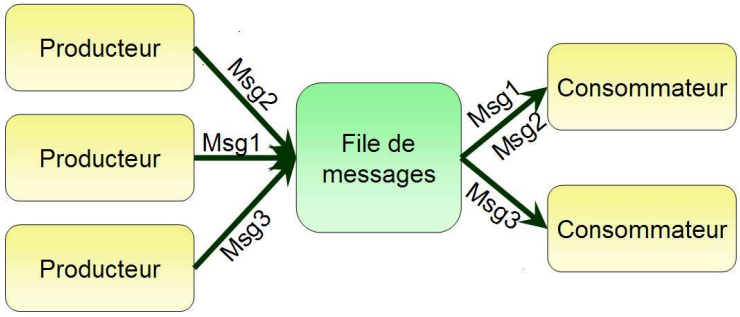


*Fig. I.5:* Principe de communication par file de messages.

d'authentification. L'utilisation de cette file de messages se déroule via des méthodes dédiées de production et de consommation de messages : le producteur soumet explicitement ses messages via une méthode invoquée sur la file, et le consommateur peut choisir soit d'invoquer une méthode pour récupérer les messages, soit de fournir une méthode *callback* permettant à la file de fournir les messages au fur et à mesure qu'ils arrivent.

La construction des messages à envoyer se fait "manuellement", c'est-à-dire qu'il est à la charge du programmeur de fournir à l'objet `Message` le contenu qu'il veut envoyer. Dans le cas d'un envoi de plusieurs objets ou données, il doit créer un tableau d'objets (ou de données) et le remplir, ou bien définir un objet possédant tous les objets comme attributs, avant de le soumettre au `Message`, qui pourra ensuite être envoyé via l'objet file de message.

### *I.4.4 Les événements*

Les événements sont une évolution du système de files de messages. En fait, ils se basent sur le même principe de producteur et de consommateur. La principale évolution entre les deux systèmes est que les événements s'intègrent à la programmation par implémentation d'une interface. Il est possible de produire et consommer les événements selon deux comportements distincts : le *push* et le *pull*. Dans le premier cas, le producteur déclenche ses événements directement sur le canal de communication (i.e. explicitement), et la méthode dédiée est appelée chez le consommateur pour lui soumettre l'événement. Dans le second cas, le fonctionnement est inverse : lorsque le consommateur veut obtenir un événement, il soumet la requête explicitement sur le canal de communication, et la méthode dédiée est appelée chez le producteur pour qu'il soumette un événement.

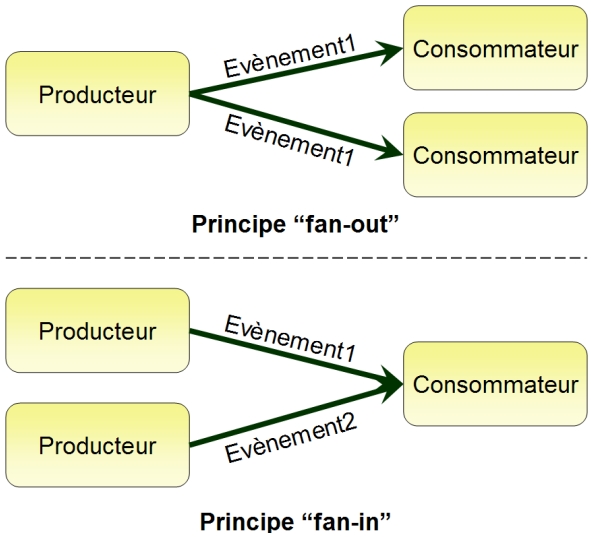


*Fig. I.6:* Principe de communication par événements.

Il est intéressant de noter que dans le fonctionnement *pull*, le modèle de programmation obtenu est de type synchrone : l'appel de méthode par le consommateur est bloquant, il déclenche une requête immédiate chez le producteur, et la réponse est obtenue directement.

Comme les files de messages, les canaux de communication d'événements permettent à plusieurs producteurs d'émettre des événements sur le même canal (communication de type *fan-in*), et à plusieurs consommateurs de recevoir ces mêmes événements (type *fan-out*).

### *I.4.5 Les sujets d'intérêt, "publish/subscribe"*

Les modèles de programmation basés sur les sujets d'intérêt présentent un fonctionnement similaire aux files de messages et aux événements, tout en possédant quelques spécificités. Lorsqu'un composant publie un message sur un sujet d'intérêt, ce dernier est acheminé vers tous les souscripteurs de ce sujet à cet instant. Comme les événements, l'intérêt est ici la réplication du message, pour que chaque souscripteur en possède un exemplaire. la persistance dans ce modèle de programmation est moins forte. En effet, si un composant souscrit à un sujet après la publication d'un message, il ne pourra pas le récupérer.

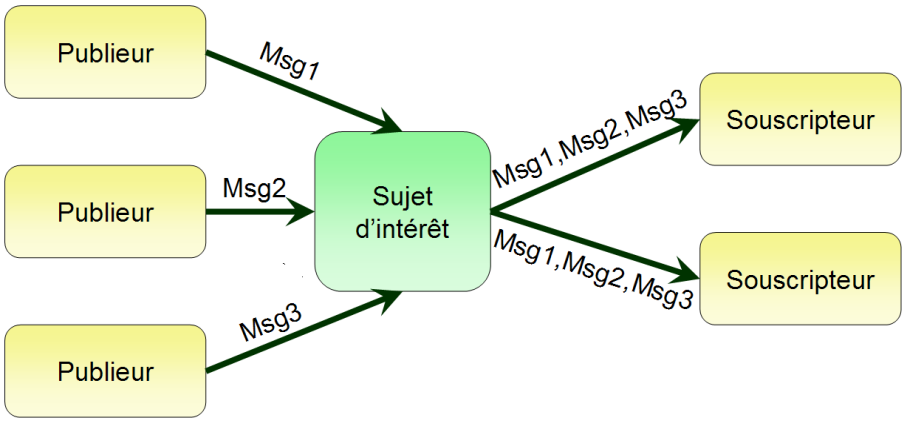


*Fig. I.7:* Principe de communication par sujets d'intérêt.

Le modèle de sujet d'intérêt peut être considéré comme un modèle d'événements moins couplé : aucun mécanisme correspondant au "pull" du modèle d'événements n'existe ici ; les producteurs et consommateurs n'ont aucune connaissance les uns des autres.

### *I.4.6 Les espaces partagés*

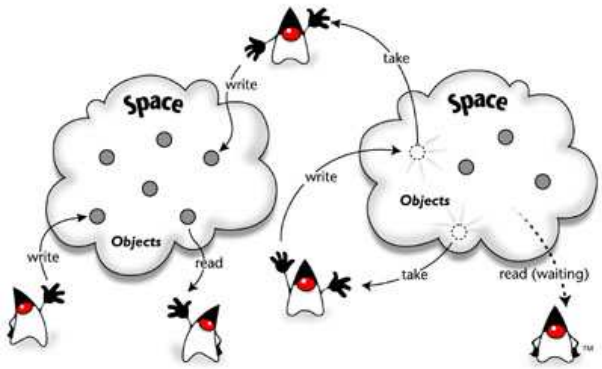


*Fig. I.8:* Principe d'espaces partagés : JavaSpaces.
©Sun Microsystems

Les modèles de programmation proposant un système d'espaces partagés fournissent principalement un objet dédié "Espace Partagé", sur lequel chaque composant peut demander de déposer, consulter ou retirer un objet

(ou un tuple, selon le modèle). Cet objet espace représente une partie du contenu d'un (ou plusieurs) serveur de stockage, et le modèle de programmation doit assurer sa cohérence : deux composants ne peuvent retirer un même objet de l'espace partagé.

Le choix de l'objet à retirer ou consulter peut se faire simplement (par correspondance de classe), ou par le biais d'une requête chargée de sélectionner le ou les objets répondant à certains critères.

A l'instar des files de messages et sujets d'intérêt, les espaces partagés peuvent être réservés à certains composants, ne possédant pas les mêmes droits sur l'espace. Un système d'authentification permet à chaque composant de s'identifier auprès de l'espace partagé afin de lui autoriser l'exécution de certaines actions définies par le créateur de l'espace.

### *I.4.7 Synthèse*

Si les modèles de programmation synchrones de type RPC sont mieux intégrés au langage objet et donc plus faciles à mettre en œuvre, le principal intérêt des modèles asynchrones est que tous gèrent les communications à partir et vers plusieurs composants. De plus, ils permettent d'établir des communications sans pour autant connaître les autres composants qui y prennent part. Néanmoins, le modèle RPC reste le moyen le plus simple pour établir des communications bidirectionnelles entre composants (le lien retour dans une communication asynchrone n'est jamais trivial).

Les différents modèles de programmation existants semblent couvrir tous les besoins des développeurs en matière de fonctionnalités et de propriétés des communications (synchronisme ou asynchronisme, couplage fort ou faible), fournissant même des patrons de conception aidant au fonctionnement interne de l'application (méthodes callback, et paradigmes fournis par les modèles asynchrones). Si aucun nouveau concept de modèle de programmation répartie ne semble devoir être conçu, il manque toutefois un modèle qui fournirait une combinaison de plusieurs des paradigmes cités plus haut.

## *I.5 Modèles de programmation basés sur les annotations*

Les modèles de programmation des intergiciels sont en général basés sur les bases du langage utilisé : les appels de méthode. Certains fournissent simplement des méthodes de communication (e.g. `Socket`, JMS), et d'autres

permettent d'utiliser les méthodes classiques des composants pour que la communication se fasse implicitement.

Certains intergiciels ont toutefois pris le parti de proposer un modèle de programmation particulier, basé sur une notion moins mêlée au code de l'application : les annotations. Leur utilisation peut être exclusive ou en complément d'un modèle plus traditionnel, mais les intérêts vantés par tous sont la clarté et la simplicité. En effet, les annotations sont distinctes du code ; elles ne sont pas situées dans d'autres fichiers, comme le seraient des descriptions ou configurations XML, mais tout en étant attachées à des éléments du code, elles s'en différencient aisément.

L'avantage certain des annotations face aux aspects [KLM+97] est leur compatibilité et leur intégration au langage Java, qui les rendent portables et utilisables à différents stades : elles peuvent être interprétées directement dans le code source (comme le propose Spoon [Paw06]), ou bien lors d'une modification "à froid" du code compilé, ou au chargement de l'application (au *loadtime*), ou enfin "à chaud" pendant l'exécution de celle-ci (au *runtime*, tout objet Java peut consulter les annotations présentes). Si ces annotations ne sont pas interprétées, cela n'a aucune incidence sur l'exécution de l'application ; les annotations sont simplement ignorées.

### *I.5.1 Web Services par les annotations Java*

Java propose une interface de programmation d'application (API) dédiée aux Web Services nommée JAX-WS [Mic06b] (pour Java API for XML based Web Services, anciennement JAX-RPC), et utilisant les annotations comme descripteur pour établir la distribution des applications. Les annotations utilisées par JAX-WS sont en fait issues de plusieurs spécifications distinctes (JSR 181 : Web Services Metadata ; JSR 224 : JAX-WS ; JSR 222 : JAXB ; JSR 250 : Annotations communes).

### *I.5.2 Entreprise Java Beans (EJB) par les annotations*

Les annotations sont liées depuis quelques années déjà à JEE (anciennement J2EE), notamment au travers du projet BeeHive d'Apache [SOS+06], visant à définir des annotations pour simplifier la programmation de Java Entreprise Edition.

Aujourd'hui une partie de JEE présente un modèle de programmation par annotations : les Entreprise Java Beans dans leur version 3 [Mic06a]. Les

possibilités offertes par ces annotations sont assez nombreuses, elles peuvent spécifier les interfaces de description de service, les classes et objet offrant des services, ainsi que les connections (*bindings*) avec des services distants (grâce à l'annotation `@Reference` qui spécifie le nom du service).

### *I.5.3 SCA par annotations*

L'architecture de composants de services [Cha07] propose un modèle de programmation à base d'annotations Java, utilisable seul ou en complément du modèle plus classique. Les annotations fournies présentent une utilisation similaire au modèle des EJB3.

Une application peut donc être entièrement distribuée grâce à ces annotations spécifiques. L'intérêt supplémentaire de ces annotations, et qu'elles sont vouées à être compatibles avec le modèle de programmation par annotations des EJB et des Web Services (certaines sont d'ailleurs identiques aux annotations définies par les EJB 3).

### *I.5.4 Synthèse*

Plusieurs modèles de programmation d'intergiciels proposent l'utilisation d'annotations pour spécifier la répartition de l'application. Il apparait cependant que ces annotations ont, jusqu'à maintenant, toujours besoin d'un autre modèle de programmation complémentaire.

## *I.6 Synthèse de l'état de l'art*

De très nombreuses possibilités sont offertes aux développeurs souhaitant concevoir une application répartie. Mais le dénominateur commun reste toujours l'intergiciel. Il permet de s'abstraire de la couche réelle de communication, et introduit des concepts de plus haut niveau comme les services, les objets et composants répartis, les files de messages, les espaces partagés.

Les intergiciels les plus utilisés aujourd'hui présentent un modèle de programmation fortement couplé à un modèle de communication. Ils ne proposent pas ou peu de possibilités d'adaptation aux changements de contexte. Ils sont donc très efficaces dans un contexte donné, mais leur conception statique pose vite problème dans des situations de mobilité par exemple.

Il existe également des intergiciels réflexifs ou multi-modèles, capables d'une part de modifier leur propre comportement pour s'adapter aux changements extérieurs, et d'autre part de communiquer avec des composants développés avec plusieurs intergiciels différents.

Les annotations offertes par le langage de programmation Java permettent d'abstraire certaines notions du code applicatif. Plusieurs intergiciels actuels proposent un modèle de programmation présentant ces annotations, ce qui permet de séparer clairement le code applicatif du code de répartition.

| | SCA | Flexinet | PolyORB | DynamicTAO | Jonathan | ReMMoC | Commutation S/A |
|---|---|---|---|---|---|---|---|
| Prog. Synchrone | ● | ● | ● | ● | ● | ● | ● |
| Prog. Asynchrone | ● | | ● | | | | ● |
| Patrons de conception | | | | | | | |
| RMI / CORBA | ● | ● | ● | ● | ● | ● | ● |
| Web Services | ● | | | | | ● | |
| JMS | ● | | ● | | | | ● |
| JavaSpaces | | | | | | | |
| Intéropérabilité | ● | | ● | | ● | ● | |
| Adaptation dynamique | | ● | | ● | | | ● |

*Fig. I.9:* Propriétés multi-modèles des différents intergiciels présentés.

Il n'existe pour le moment aucune "solution miracle" capable de fournir un modèle de programmation complet et performant, un ou plusieurs modèles de communication offrant interopérabilité et efficacité, et des mécanismes d'adaptation pour prendre en compte les changements de contexte pour changer son comportement. SCA semble être très prometteur en spécifications, et les premières implémentations tiennent certaines de ces promesses jusqu'ici. Il semble être l'intergiciel le plus proche de cette "solution miracle", mais le manque d'adaptation et de reconfiguration est une réelle lacune pour une utilisation en mobilité.

## *I.7 Positionnement de nos travaux*

Les travaux présentés dans ce mémoire s'appuient sur l'état de l'art exposé ici, et en particulier sur les lacunes des systèmes actuellement proposés. Certaines notions bien établies sont pertinentes, d'autres ne semblent pas assez poussées, et enfin des notions particulièrement importantes à nos yeux ne sont tout simplement pas proposées.

Au regard de ce panorama actuel des intergiciels de communication, nous allons donc positionner notre approche et nos travaux face aux propositions existantes. Les détails des notions traitées ici seront évidemment développés dans les chapitres suivants.

### *I.7.1 Utilisation des annotations*

Le modèle de programmation que nous proposons utilise, entre autres, des annotations. Les projets actuels tendent à établir ces annotations comme standard pour spécifier des propriétés et mécanismes indépendants du code métier. Nous permettons ainsi d'abstraire le code de répartition du code de l'application pour clarifier son développement.

### *I.7.2 Couplage au sein de l'intergiciel*

Les intergiciels présentent en général un fort couplage entre le modèle de programmation utilisé, et le modèle de communication sousjacent. Quelques projets comme PolyORB ou ReMMoC proposent de découpler ces modèles pour offrir une plus grande modularité dans le développement et le déploiement des applications. Nous faisons également ce constat dans notre approche, et poussons même cette notion plus loin en **découplant** d'une part **le modèle de programmation du modèle de communication**, mais également en **découplant le modèle de programmation du client de celui du serveur**. L'intérêt de ce principe sera détaillé par la suite.

### *I.7.3 Comportement réflexif / multi-modèles*

Les intergiciels présentant des propriétés de comportement réflexif ou bien d'utilisation de modèles de communication multiples ne sont pas très nombreux, pourtant leur approche est justifiée et pertinente. Nous prenons également cette voie dans notre approche : notre proposition intègre d'une part des mécanismes de reconfiguration dynamique et de prise en compte du contexte, et elle met en œuvre d'autre part plusieurs modèles de communication. Ces modèles multiples offre ainsi une certaine interopérabilité (comme les projets multi-modèles existants), mais également une capacité d'adaptation aux conditions d'exécution de l'application, ce qui représente l'un des points clés de nos travaux.

# II

# Approche multi-modèles

Les travaux de cette thèse sont basés sur une approche multi-modèles. Cette approche propose de combiner plusieurs modèles de programmation et de communication afin d'obtenir une application capable d'utiliser simultanément plusieurs types de programmation, et pouvant communiquer avec des composants répartis de plusieurs types. Cette approche est le fruit de certains constats concernant l'utilisation et le déploiement des intergiciels actuels, et s'inspire d'approches similaires développées dans certains projets (cf section I.3) en proposant de plus des concepts originaux et novateurs.

## *II.1 Constats*

Les intergiciels "classiques"[1] actuels présentent de nombreux points communs dans leur approche et dans les services qu'ils fournissent, comme le détaille le chapitre I. Le développeur utilise un modèle de programmation (aussi appelé "personnalité applicative" dans d'autres approches comme PolyORB [Qui03]) plus ou moins évident et complet, afin de permettre la communication entre composants répartis dans son application par le biais d'un modèle de communication (appelé également "personnalité protocolaire").

Cette approche classique est un premier pas pour faciliter le développement d'applications réparties, mais limite l'hétérogénéité des paradigmes de

[1] Par intergiciel "classique" ou "conventionnel" nous désignons les intergiciels mono-modèles comme définis au chapitre I

communication dans une même application, et ne permet pas ou peu de facultés d'adaptation. Certains points particuliers de la programmation et du déploiement d'applications réparties nous amènent à constater les limites induites par ces intergiciels "conventionnels".

### *II.1.1 Modèles de programmations complémentaires*

Différents types de programmation existent pour développer des applications réparties, comme cela a été développé au chapitre précédent. Les modèles de programmation synchrones d'invocation distante sont adaptés à la plupart des développements, et c'est la raison pour laquelle la majorité des intergiciels actuels le proposent : la programmation répartie, pour être transparente, doit s'intégrer à la programmation de l'application, donc au langage procédural. Cependant, le paradigme de programmation RPC n'est pas forcément pertinent devant toutes les situations ; e.g. les systèmes d'espaces partagés sont plus à-même de répondre aux besoins de programmation spécifiques aux développements sur grilles de calcul [YB04].

Chaque modèle de programmation répartie est adapté à une ou plusieurs situations de développement précises, mais aucun modèle aujourd'hui ne peut se vanter de répondre le mieux à tous les cas existants de communication entre composants répartis. Certains [CT94] arguent qu'un modèle de type RPC peut être utilisé pour résoudre tous les cas. C'est en effet le cas : le RPC étant une extension du langage procédural, si l'on a besoin d'une file de message, il suffit de développer un objet intégrant les mécanismes d'une file de messages, et de le rendre accessible via RPC. Cependant, l'utilisation ici d'un modèle de programmation basé directement sur les files de messages présentera deux avantages :

- Le développeur n'aura pas besoin de développer ses mécanismes de file de messages, le modèle de programmation lui les fournit. Le développement s'en voit optimisé.
- Le choix du "lieu d'implantation" de la file d'attente n'a plus lieu d'être. En effet, en utilisant le RPC le développeur doit choisir si la file d'attente doit être instanciée par le producteur, le consommateur, ou un composant tiers. En utilisant un modèle de file de messages, l'intergiciel gère lui-même le lieu d'implantation, et les spécificités de communication sous-jacentes (comme la temporisation des communications si la file n'est pas joignable).

Le modèle de programmation fourni par un intergiciel de communication est le reflet du paradigme de communication proposé, et les mécanismes in-

ternes de cet intergiciel ainsi que le modèle de communication utilisé sont choisis et adaptés en conséquence : e.g. une programmation synchrone de type RPC exigera une chaine de communication, et par conséquent un modèle de communication synchrone et performant. La foultitude de modèles de programmation proposés actuellement par les différents intergiciels permettent l'utilisation de paradigmes variés, et chaque type de paradigme gagne à être utilisé dans certains cas. Cette multiplicité de modèles permet aux développeurs de choisir l'outil de communication le plus adapté à leurs besoins, en fonction des impératifs posés par le type de composants répartis et au type d'interactions qu'ils présentent. Ces différentes familles de modèles de programmation sont ainsi complémentaires plutôt que concurrentes.

### *II.1.2 Hétérogénéité des modèles de communication*

Le développement d'applications réparties peut se confronter à deux situations particulières :

- L'application ne dépend d'aucun composant actuellement existant ; elle se suffira à elle-même en termes de fonctionnalités. Les composants de cette application seront tous développés pour celle-ci, sans aucune réutilisation de l'existant. Dans ce cas, le problème principal se situe au niveau du choix du modèle de communication à utiliser ; il devra permettre à l'application d'assurer sa tâche au mieux (e.g. un modèle direct et propriétaire comme celui de .Net Remoting sur TCP privilégiera les performances, le modèle SOAP sur HTTP permettra plus d'interopérabilité).
- L'application devra se greffer sur des composants existants, ou utilisera des services répartis existants. Le choix du modèle de communication est alors calqué sur le modèle déjà utilisé par ces composants ou services. Le choix devient plus compliqué lorsque les composants présentent des modèles distincts.

L'hétérogénéité des modèles de communication est inévitable, ceci étant dû à des impératifs de réutilisation de l'existant ou à des choix techniques liés au déploiement. Lorsqu'une application est confrontée à une multiplicité des modèles utilisés, plusieurs choix s'offrent aux développeurs.
Dans un premier temps, si cela est possible, plusieurs intergiciels peuvent être utilisés lors du développement, afin de pouvoir interagir via les différents modèles. Cela demande tout d'abord d'utiliser le modèle de programmation de ceux-ci simultanément (et donc de les maitriser), et ensuite de charger les différents gestionnaires lors de l'exécution de l'application, ce qui peut engendrer une certaine complexité de configuration et de déploiement, et une

empreinte mémoire relativement conséquente.
Dans un deuxième temps, des passerelles entre modèles de communication [Bak01] peuvent être développées et utilisées, afin de permettre la communication avec les composants n'utilisant pas le modèle retenu pour le développement de l'application. Ces passerelles doivent être conçues spécifiquement pour chaque couple modèle source/modèles cible [QKP99], ce qui représente un travail de développement supplémentaire non négligeable.

### *II.1.3 Modèles de communication complémentaires*

Les différents modèles de communication présentent des caractéristiques différentes, tant au niveau des performances que de la sécurité, et de la tolérance aux erreurs. Chaque composant réparti, s'il a été bien pensé et conçu, doit utiliser le modèle de communication le mieux adapté à son utilisation, privilégiant soit les performances, soit la capacité à être accessible[1]. Mais quid d'un composant dont l'utilisation n'est pas définie lors de son développement, ou dont le contexte de déploiement est susceptible de changer ? Pouvoir utiliser le modèle de communication le plus adapté à la situation courante semble être une nécessité dans ce type de contexte, et si cette situation change, il est alors nécessaire de changer le modèle utilisé en conséquence.

La complémentarité des modèles de communication ne fait aucun doute, aux vues des choix proposés par les intergiciels actuels : .Net Remoting propose deux protocoles (SOAP et propriétaire)sur deux moyens de communication distincts (TCP et HTTP), RMI fait de même avec JRMP et IIOP. Ce principe de multiplicité des modèles a deux objectifs qui sont l'adaptation aux modèles existants, et l'adaptation au contexte d'exécution et de communication du composant. Ce dernier objectif pourtant primordial n'est cependant pas exploité par les principaux intergiciels multi-modèles actuels.

### *II.1.4 Noyau commun aux différents types d'intergiciel*

Comme plusieurs projets l'ont démontré [QKP01, BCA+01], les différents intergiciels existants sont basés sur des mécanismes internes similaires, qui peuvent être factorisés au sein d'un noyau commun, sur lequel viennent se greffer les services spécifiques à chaque modèle de programmation et de communication.

[1] E.g. un composant offrant un modèle de communication sur HTTP restera accessible derrière des pares-feu, ce qui ne sera sans doute pas le cas avec un modèle RMI.

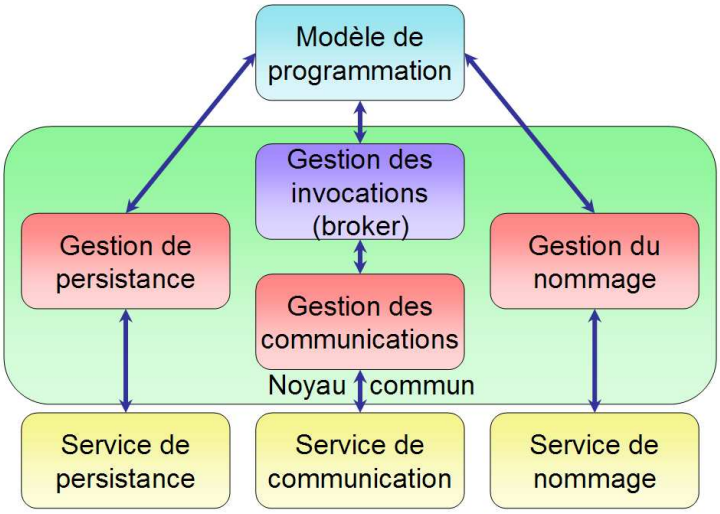


*Fig. II.1:* Architecture d'un intergiciel mettant en évidence le noyau commun.

Que les intergiciels proposent un paradigme de programmation et de communication synchrone, asynchrone ou autre, les services basiques fournis se reposent sur les mêmes principes, et donc une partie de leur fonctionnement est similaire. L'architecture interne d'un intergiciel peut être schématisée en trois parties [Pau02], qui sont :

- Le modèle de programmation, autrement appelé la personnalité applicative. C'est l'interface entre le développeur (i.e. les objets développés) et l'intergiciel.
- Le noyau commun, aussi désigné par le terme de couche neutre. C'est le ou les composants rassemblant les mécanismes génériques à tout intergiciel.
- Le modèle de communication, autrement appelé la personnalité protocolaire. C'est le moyen de communication entre les noyaux communs.

## *II.2 Principes de l'approche multi-modèles*

En se basant sur les constats que nous venons d'énoncer, nous proposons une approche multi-modèles pour résoudre les problèmes qui persistent encore aujourd'hui dans l'utilisation des intergiciels de communication en général, et en particulier en contexte de mobilité.
D'autres projets proposent d'ores et déjà une approche basée sur certains de ces constats, et tous s'accordent à penser et à prouver qu'un solution multi-modèles semble être la meilleure optique à adopter.

Notre approche multi-modèles se décompose en trois points, dont deux sont issus d'un raisonnement plus abouti que ce que proposent les projets actuels. Tout d'abord, un modèle de programmation complet et proposant un panel exhaustif des paradigmes et patterns de programmation répartie existants, puis un ensemble de modèles de communication, offrant les différents protocoles et supports de communications actuels utilisables séparément ou en combinaison, et enfin un noyau commun intégrant tous les mécanismes récurrents des différents intergiciels, et présentant des composants de programmation, de communication, et un noyau de répartition générique. Ce dernier point est une notion déjà bien établie dans la littérature, nous n'en détaillerons donc pas le principe, mais nous en reparlerons plus en détails pour expliquer son application dans l'architecture ainsi que son implémentation concrète.

### *II.2.1 Modèle de programmation complet*

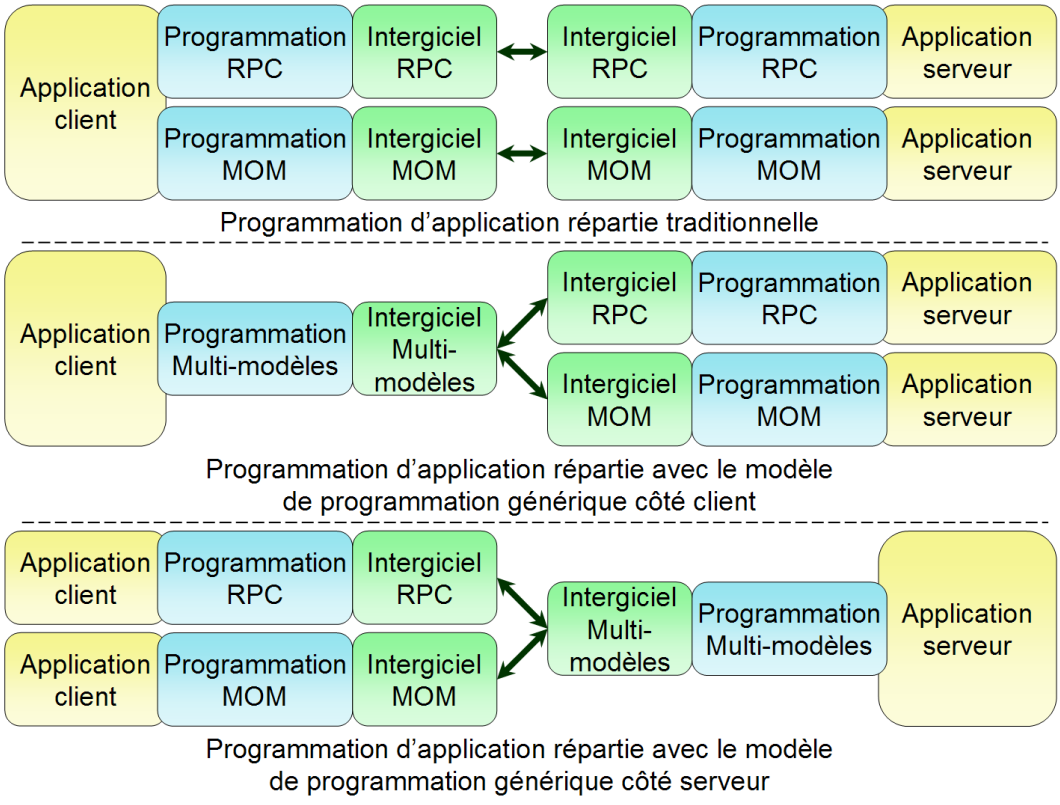


*Fig. II.2:* Modèle de programmation étendu proposé par l'approche multi-modèles

Pour permettre au développeur d'une application d'optimiser le temps de programmation, il est nécessaire de lui fournir tous les outils possibles, en particulier des moyens simples de faire communiquer ses composants logiciels, ainsi que des patrons de conception permettant de mettre en œuvre les différents paradigmes actuels de programmation répartie. En effet, les intergiciels actuels fournissent un modèle de programmation permettant un type de programmation cohérent avec le modèle de communication utilisé, mais cela interdit au développeur d'utiliser les outils de conception offerts par les autres modèles de programmation. Ainsi lors d'un développement à l'aide de Java RMI, si l'utilisation d'un espace partagé permet de simplifier les développements et optimise les communications entre les composants, le développeur doit dans un premier temps concevoir un objet implémentant ce principe d'espace partagé. Dans un deuxième temps, rien ne peut assurer que le modèle de communication soit bien exploité pour les communications induites par cet objet.

Pour rendre le développement plus optimisé, et l'utilisation des modèles de communication plus efficace, notre approche propose de recourir à un modèle de programmation complet, présentant d'abord un modèle de programmation synchrone et asynchrone, et ensuite des patrons de conception permettant de programmer efficacement à l'aide d'outils offerts par les intergiciels existants, (e.g. espaces partagés, files de messages, événements).

L'apprentissage par les développeurs d'un nouveau modèle de programmation peut être long, fastidieux, voire infructueux lorsque ce modèle présente trop de paramètres et de spécificités à gérer. L'approche multi-modèles ne peut être acceptée et reconnue que si les capacités offertes sont facilement utilisables, le modèle de programmation doit par conséquent se montrer le plus simple et performant possible.

### *II.2.2 Modèles de communication multiples*

Comme nous l'avons justifié plus haut, les services et composants répartis que requiert une application ne fournissent pas forcément tous le même modèle de communication, et certains impératifs de déploiement obligent à opter pour tel modèle plutôt qu'un autre. L'application développée devra donc potentiellement dialoguer avec des composants présentant des modèles de communication variés, et incompatibles[1] dans la plupart des cas.

[1] Certains intergiciels proposent le choix entre plusieurs modèles de communication, mais ce n'est pas le cas pour tous

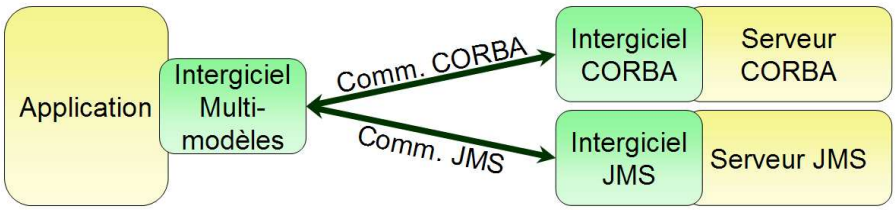


*Fig. II.3:* Communication en contexte hétérogène avec l'approche multi-modèles

Notre approche multi-modèles propose en conséquence de pouvoir utiliser un ou plusieurs modèles de communication, indépendamment du modèle de programmation choisi au développement. Cette utilisation conjointe permet de pouvoir utiliser des services et composants répartis de différents types sans pour autant modifier le type de développement de l'application.

Si certains intergiciels actuels proposent déjà l'utilisation conjointe de deux modèles de communication (e.g. RMI avec JRMP et IIOP, SOAP avec JMS et HTTP), les capacités d'interopérabilité sont néanmoins limitées. Nous proposons d'utiliser conjointement des modèles de types différents (synchrone, asynchrone, ou autre) pour communiquer avec des applications utilisant nativement l'intergiciel correspondant au modèle de communication.

### *II.2.3 Combinaison des modèles de communication*

Ce type d'utilisation simultanée de plusieurs modèles de communication a déjà été exploré et validé par d'autres recherches [GBS03, Cha07], notre approche présente cependant de nouvelles perspectives quant aux capacités d'adaptation au contexte des applications réparties.
En effet, nous considérons l'utilisation de composants présentant différents modèles, mais également et surtout l'utilisation de composants eux-mêmes multi-modèles. Les possibilités dans ce cas sont bien plus poussées en terme d'adaptation et de choix de modèle.

Chaque modèle de communication présente ses points forts et points faibles selon son contexte de déploiement, la faculté pour des composants répartis de pouvoir utiliser plusieurs modèles distincts augmente d'autant leur faculté à fournir des communications optimales dans différents contextes de déploiement.
Un composant avec lequel toute application peut communiquer via un modèle Web Service, mais également via des modèles CORBA et JavaSpaces a

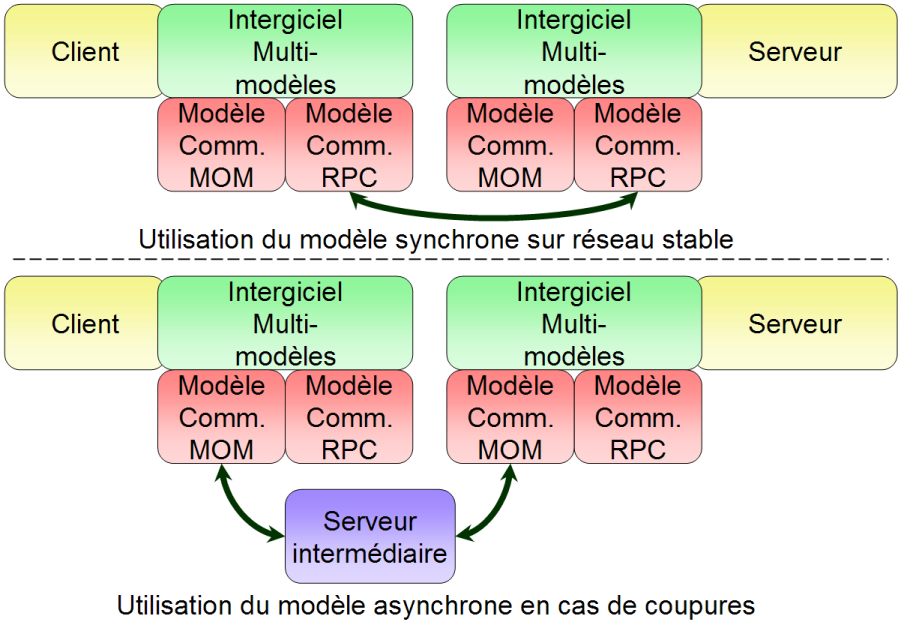


*Fig. II.4:* Exemple de combinaison des modèles de communication selon l'approche multi-modèles

un intérêt certain. Avec les outils de communication actuels, un tel composant demande de développer une interface pour chaque modèle fourni, alors que notre approche multi-modèles le permet automatiquement.

Le cas d'application le plus parlant de notre approche est sans nul doute une situation où l'application et le composants offrent un comportement multi-modèles. Ceux-ci ont alors le "choix" du modèle qu'ils vont utiliser pour communiquer. Si jusqu'alors l'approche permettait une adaptation "subie" en fournissant un unique modèle de communication possible, il est possible ici d'opter pour plusieurs modèles combinés. Nous parlerons ici d'adaptation "choisie", car en plus de la sélection de modèles effectuée d'office par les circonstances (i.e. seuls les modèles offerts en commun par l'application et le composant réparti sont retenus), un choix plus précis peut être fait en fonction du contexte afin d'optimiser les performances et la sûreté de la communication.

### *II.2.4 Verrous et challenge*

Notre approche multi-modèles présente des concepts novateurs capables de faciliter et d'optimiser le développement et le déploiement des applications

réparties, en leur permettant de s'adapter dynamiquement au contexte. Cependant, l'application de cette approche représente un challenge car elle se heurte à plusieurs verrous technologiques :

Premièrement, le modèle de programmation doit offrir tous les paradigmes et les patrons que le développement d'applications réparties optimisées nécessitera, tout en restant le plus simple, accessible et compréhensible possible. En effet, un outil rapide et facile à s'approprier sera plus aisément accepté et donc exploité. Or il est fréquent de voir les modèles de programmation se complexifier à mesure de l'ajout de fonctionnalités.

Deuxièmement, la rupture des liens associant les modèles de programmation et de communication ne peut être faite qu'après avoir défini clairement le noyau commun. Ce dernier sera un intermédiaire neutre entre les couches applicative et protocolaire, et permettra ainsi d'associer toute personnalité de ces deux couches.

Troisièmement, le bien fondé de notre approche demeure justifié si le changement de modèle de programmation ou de communication reste totalement transparent. Les mécanismes offerts par le modèle de programmation doivent par conséquent exploiter le mieux possible le ou les modèles de communication utilisés. Il est évident que tous les concepts de programmation répartie fournis par le modèle de programmation devront être utilisables via tous les modèles de communication fournis. En plus d'être neutre, la couche intermédiaire devra donc être optimisée ; le noyau devra être assez complet pour permettre tous les concepts, mais assez concis pour éviter de présenter des fonctionnalités spécifiques à un modèle en particulier.

## *II.3 Inspiration des concepts présents dans l'état de l'art et innovations*

Notre approche, bien que novatrice, s'inspire pour une part des notions présentées et justifiées dans l'état de l'art. Nous allons donc expliquer ici ce qui nous a semblé pertinent, ainsi que les concepts originaux que nous introduisons dans nos travaux.

### *II.3.1 Modèle de programmation générique*

Dans un premier temps, un certain nombre de travaux connexes argumentent sur l'intérêt d'un modèle de programmation générique, e.g. Poly-

ORB, plus ouvert que ceux fournis par les intergiciels standards actuels. C'est également notre point de vue, mais nous enrichissons le principe en proposant en complément des patrons de conception intégrés à l'infrastructure, permettant des communications particulières entre les objets.

De plus, là où certains intergiciels proposent des mécanismes asynchrones via des appels unidirectionnels ou à l'aide de méthodes "*callback*", nous proposons un mécanisme d'optimisation automatique du code lors d'un appel asynchrone, cf III.2.4.

### *II.3.2 interopérabilité*

Dans un deuxième temps, plusieurs intergiciels multi-modèles tels que SCA et ReMMoC sont capables d'utiliser les différents modèles afin de communiquer avec des applications développées grâce à un intergiciel standard (e.g. CORBA ou Web Service). Notre approche en est également capable, mais propose des communications avec des applications standard client **et** serveur, (e.g. là où ReMMoC n'est que client).

### *II.3.3 Combinaison de modèles pour l'adaptation*

L'utilisation conjointe de plusieurs modèles de communication permet, en plus de l'interopérabilité, de s'adapter au contexte en utilisant le modèle le plus pertinent. C'est dans un troisième temps l'approche développée par V. Budau et al. et dans FlexiNet. Notre approche introduit en complément la capacité de changer de modèle de communication au cours d'une même invocation (i.e. l'émission et la réception ne passent pas par le même modèle). De plus, les mécanismes d'adaptation dynamique que notre approche présente permettent de changer le choix du ou des modèles à utiliser en fonction du contexte, ainsi que les règles dirigeant ce choix, et les observations sur lesquelles s'appuient ces règles.

### *II.3.4 Découplage des modèles de programmation et de communication*

Les intergiciels récents comme SCA commencent à souligner le fait que le fort couplage entre un modèle de programmation et un modèle de communication n'est pas forcément justifié et pertinent. C'est dans un quatrième temps ce que nous soulignons également. Mais outre ce découplage modèle de programmation / modèle de communication, nous proposons également un découplage entre le modèle de programmation utilisé par le client et par le serveur, rendant le système encore plus modulable.

## *II.4 Synthèse de notre approche multi-modèles*

De l'état de l'art actuel concernant les intergiciels, nous tirons quelques conclusions et constats : les différents modèles de programmation proposés dans la littérature et dans les intergiciels utilisés de nos jours présentent chacun un intérêt propre, optimisant le développement d'un certain type d'applications. Les modèles de communication proposés sont multiples et variés, ce qui pose un problème d'interopérabilité lorsque l'on est confronté à des communications hétérogènes au sein d'une même application. Ils présentent de plus des propriétés spécifiques qui destinent chaque modèle à un contexte particulier. Enfin, le fonctionnement interne d'un intergiciel s'appuie sur des services de base que l'on retrouve dans tous les intergiciels existants.

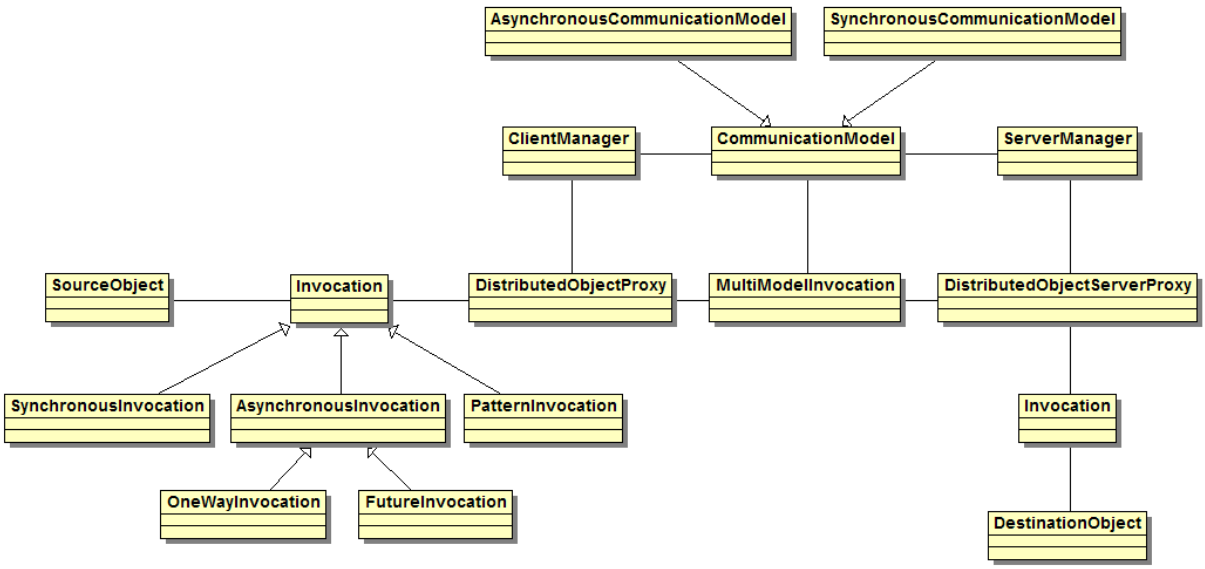


*Fig. II.5:* Métamodèle d'une invocation par une approche multi-modèles.

Nous proposons donc notre approche multi-modèles : un modèle de programmation générique permet la réunion des différents modèles existants, et autorise ainsi le développeur à utiliser le ou les modèles qui seront le plus adapté à sa technique de programmation. Puis une utilisation conjointe de plusieurs modèles de communication permet quant à elle d'assurer l'interopérabilité avec des applications existantes développées à l'aide d'intergiciels variés. Enfin ces mêmes modèles de communication combinés permettent

d'assurer des communications sûres, fiables et performantes entre applications présentant cette approche.

# III

# Infrastructure logicielle multi-modèles

Pour valider et appliquer notre approche multi-modèles, nous l'avons mise en œuvre au travers d'une infrastructure logicielle multi-modèles, que l'on pourrait qualifier également d'intergiciel combinant plusieurs personnalités applicatives et protocolaires.

Cette infrastructure vise d'une part à simplifier et aider au développement d'applications réparties, et d'autre part à permettre aux applications ainsi développées une bonne interopérabilité avec les autres composants répartis, ainsi qu'une faculté d'adaptation au contexte grâce à l'utilisation simultanée de plusieurs modèles de communication.

Tout d'abord, nous allons présenter son architecture globale, i.e. sa structure centralisée et ses composants principaux. Puis nous allons présenter précisément le rôle joué par chacun de ces composants, son comportement et son fonctionnement interne. Enfin, un point sera fait sur la gestion des cycles de vie des objets répartis dans notre infrastructure logicielle, et nous illustrerons la généricité de notre système en simulant à l'aide de notre infrastructure des intergiciels existants.

## *III.1 Architecture générale*

Bien que notre infrastructure logicielle multi-modèles ne soit pas basée sur un modèle de composants particulier, nous la présentons néanmoins comme

un assemblage de composants "conceptuels" (cf 1.3, représentant chacun une fonction principale de l'infrastructure. L'organisation de ces composants est détaillée ci-après, ainsi que les principes de fonctionnement de notre infrastructure, synthétisés dans son méta-modèle.

Plusieurs types d'architecture sont possibles pour développer un intergiciel, mais l'approche généralement choisie dans l'état de l'art est une approche modulaire basée sur un noyau central. Cette approche apporte une séparation des préoccupations supplémentaire à une architecture monolithique, en permettant à chaque service et composant d'avoir un rôle clairement défini.

### *III.1.1 Organisation centralisée des composants*

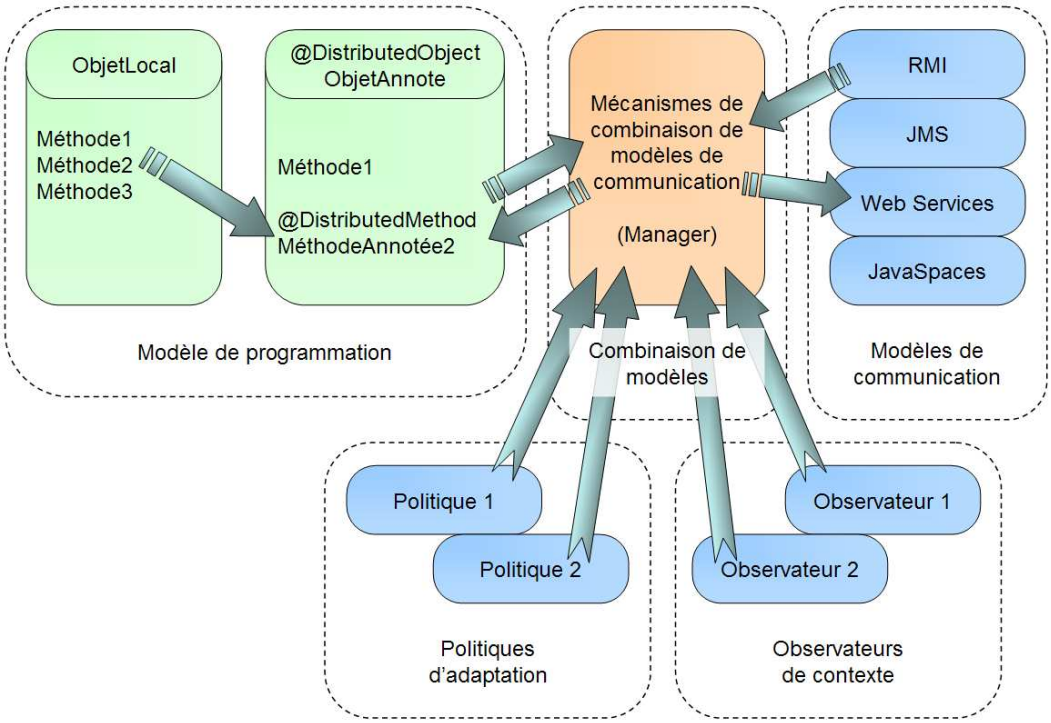


*Fig. III.1:* Architecture générale de l'infrastructure logicielle multi-modèles.

Comme il apparait sur la figure III.1, les différents composants "conceptuels" de l'infrastructure logicielle multi-modèles sont tous en relation directe avec le Manager, composant central de l'architecture pouvant être assimilé au noyau neutre cité précédemment. Cette organisation est la conséquence

des propriétés multi-modèles de notre système. En effet, pour que le fonctionnement et les interactions entre modèle de programmation et modèles de communication puissent être structurés et optimum, tout en restant cohérents et sûrs, il est nécessaire d'établir des règles, indépendantes du modèle choisi, afin de rendre le système capable d'utiliser n'importe quel modèle avec les mêmes propriétés. Le Manager assure cette constance, en permettant à tous les modèles de programmations d'utiliser aussi efficacement les différents modèles de communication sans différence notable. Il assure également le recueil des données de contexte et l'application des politiques d'adaptation.

### *III.1.2 Méta-modèle de l'infrastructure*

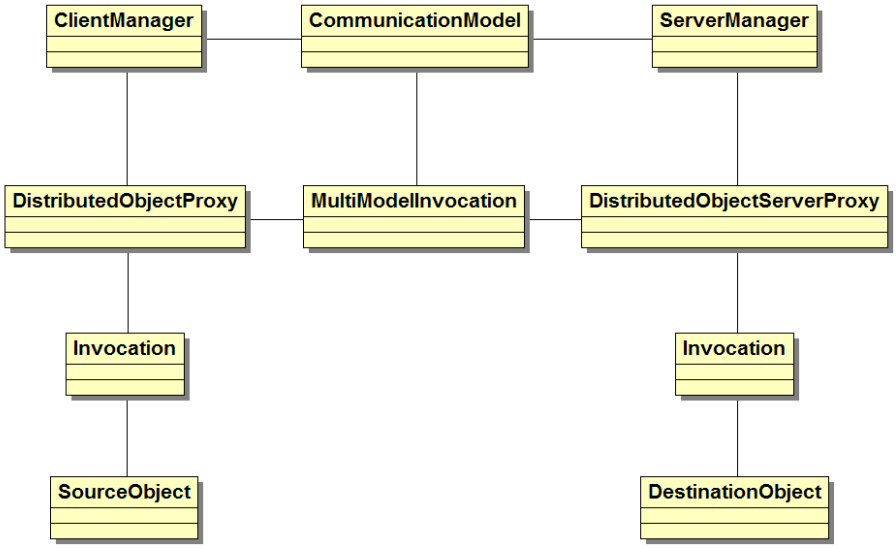


*Fig. III.2:* Méta-modèle générique d'une invocation via l'infrastructure logicielle multi-modèles.

Le principe du fonctionnement général d'une invocation via notre infrastructure logicielle multi-modèle suit le méta-modèle schématisé en figure III.2. Ce modèle résume les différents éléments de notre approche, et illustre les diverses possibilités d'utilisation de cette infrastructure pour obtenir des communications performantes, fiables et adaptées au contexte. Il prend en compte le cas particulier où les deux objets concernés par l'invocation utilisent l'infrastructure multi-modèles, afin d'illustrer le point le plus significatif de notre approche. De plus, il s'agit du cas mettant en œuvre les mécanismes d'adaptation les plus poussés.

Tout d'abord, l'objet distribué *source* de l'invocation peut initier différents types d'invocations, chacun définissant un comportement particulier. Ces types sont développés plus en détails dans la section III.2.4. Le type de l'invocation résultante au niveau de l'objet distribué *destination* de l'invocation n'est pas obligatoirement le même que celui de l'objet *source*, ce qui permet à chacun de ces objets d'adopter un comportement adapté à son exécution. E.g. un objet *source* invoque une méthode d'une façon synchrone – il nécessite obligatoirement le résultat de cette invocation pour continuer son exécution – alors que l'objet cible possède son propre fil d'exécution et gère plus efficacement les invocations en les recevant via une file de messages. Via notre infrastructure chaque objet peut ainsi conserver son mode d'exécution dédié.

Chaque objet utilise les possibilités de l'infrastructure via un objet "proxy", i.e. un intermédiaire chargé de traduire d'une part l'invocation *source* en une communication interne exploitable par les différents composants de l'infrastructure (cf section III.7), et d'autre part cette communication en invocation *cible*.
Cet objet "proxy" n'est pas explicitement mis à disposition du développeur, mais est utilisé implicitement par le modèle de programmation pour acheminer les invocations.

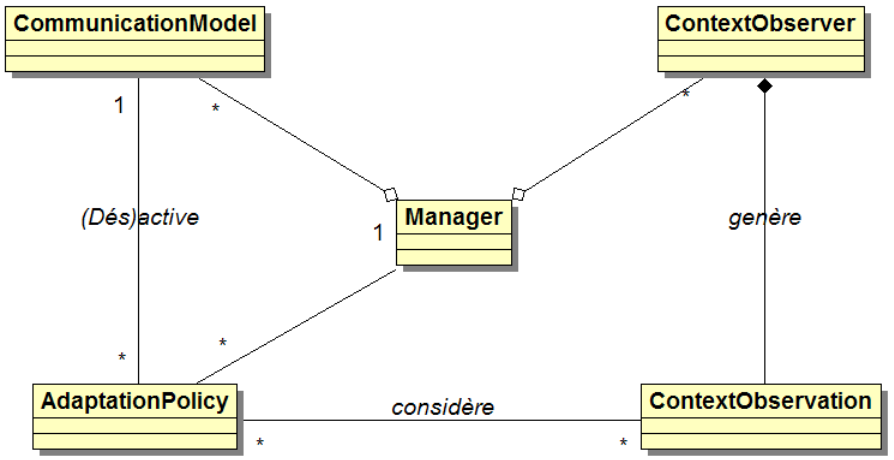


*Fig. III.3:* Méta-modèle du choix des modèles de communication en fonction du contexte dans l'infrastructure multi-modèles.

L'infrastructure (les 2 infrastructures dans le cas présenté dans le méta-modèle) se repose sur un Manager (cf section III.6) qui est chargé d'acheminer

les invocations via le où les modèles de communication qu'il aura "choisi" en fonction des politiques d'adaptation et des observations de contexte, comme le second méta-modèle le spécifie (figure III.3). Encore une fois ici le Manager occupe une place centrale dans le fonctionnement de l'infrastructure.

## *III.2 Modèle de programmation générique : la personnalité applicative*

Notre infrastructure logicielle multi-modèles apparait au développeur comme un intergiciel classique (comme CORBA) : après avoir paramétré l'environnement de l'intergiciel grâce à quelques jeux d'instructions de configuration et de composition de la répartition, le langage de programmation peut être utilisé classiquement afin d'établir les procédures de code métier. Si l'utilisation de patrons de communication spécifiques est nécessaire, l'intergiciel en fournit certains, qui sont alors utilisables comme des objets traditionnels au sein du développement.

Nous allons donc voir dans un premier temps les spécificités du modèle qui permettent au programmeur de conserver ses habitudes de développement, puis nous présenterons l'intérêt et les propriétés de l'utilisation des annotations comme outil de programmation répartie. Nous présenterons également le moyen plus explicite d'établir les communications entre objets répartis : l'appel des méthodes dédiées du Manager. Enfin nous expliquerons en détails la mise en œuvre des différents paradigmes de programmation et patrons de conception au sein de notre modèle de programmation.

### *III.2.1 Conservation de la programmation traditionnelle*

La plupart des langages de programmation présentent aujourd'hui un modèle basé sur l'appel de procédures, itérativement et récursivement. Il est important dans le cadre d'un intergiciel d'intégrer les mécanismes spécifiques à la répartition sans modifier l'utilisation originale du langage de programmation. En effet, le programmeur doit déjà intégrer de nouvelles techniques pour utiliser l'intergiciel, de plus il ne faudrait pas l'empêcher de développer les parties métier de son application comme il sait le faire, i.e. en utilisation standard du langage de programmation. Le modèle de programmation de l'intergiciel doit donc se greffer sur le langage, et non remplacer ses mécanismes.

Comme la grande majorité des intergiciels basés sur l'invocation à distance de méthodes [Mic03, GGM99], nous proposons un modèle de programmation s'intégrant au langage, en se substituant aux appels de méthodes classiques lorsque l'objet cible est distant. Pour garder également la possibilité d'accéder directement aux attributs d'un objet, même réparti, l'utilisation directe de ceux-ci est détectée lors de la modification à la volée de la classe *source*, et est alors interprétée comme un appel à des *getters et setters*[1]. Si des langages comme C#[2] permettent d'automatiser la définition de telles méthodes, nous proposons ici un système totalement transparent et automatique de génération de méthodes (si elles n'ont pas déjà été définies par le programmeur) et de modification de code pour obliger leur utilisation en accès distant.

Comme nous l'expliquons en particulier dans la section III.2.4 concernant les invocations asynchrones, le modèle de programmation que nous proposons rend également transparente l'utilisation d'appels asynchrones des méthodes, en modifiant à la volée le code pour retarder l'attente de la valeur de retour, afin de ne bloquer l'exécution du code seulement lorsque cela est nécessaire.

### *III.2.2 Annotations : séparation des préoccupations*

L'un des points les plus importants dans le développement et la maintenance de logiciels (et donc d'applications réparties en particulier) est la lisibilité et la compréhensibilité du code. Tout d'abord les langages procéduraux permettent de séparer le code en méthodes, effectuant typiquement chacune une action précise. Les langages objet permettent ensuite de regrouper ces méthodes ainsi que des attributs selon la notion qu'ils concernent. Ceci fait, le code métier est relativement clair à comprendre. Cependant des notions transverses viennent compliquer cette compréhension, en introduisant du code concernant un ensemble d'objets et de méthodes, ce qui complexifie malheureusement cette compréhension. Dans le contexte de la séparation des préoccupations (cf sectionI.3.1), la programmation par aspects [KLM+97] propose de résoudre ce problème en introduisant la notion d'aspect transverse, et permettant de programmer ces aspects séparément du code métier. Récemment, le projet Spoon [Paw05] a montré que l'utilisation des annotations pour aider à définir des préoccupations transverses est justifié et pertinente. Les intergiciels de communication proposant des annotations comme modèle de programmation (cf section I.5) font d'ailleurs leur

[1] Termes standards pour désigner les fonctions d'accès et de modification d'attributs en langage objet, respectivement de "get" (obtenir) et "set" (régler, changer, modifier).

[2] Langage objet, pilier de la plateforme .Net de Microsoft, hybride présentant des caractéristiques de C++ et de Java, ainsi que certains nouveaux outils de programmation.

apparition depuis quelques années.

Nous proposons donc un modèle de programmation basé sur les annotations Java. En plus de cette séparation des préoccupations, lorsqu'elles ne sont pas interprétées par les mécanismes adéquats, ces annotations sont simplement ignorées par le compilateur et la machine virtuelle (i.e. l'interpréteur de code compilé). Ce deuxième aspect permet à une application développée grâce à notre infrastructure logicielle d'être utilisée seule ; elle adopte alors un comportement local, dénué de toute répartition.

Dans cette optique de portabilité et de non-modification du code compilé de l'application, les annotations font partie intégrante du code, du développement à l'exécution, mais n'engendrent aucun changement dans l'application, sauf lorsque celle-ci est exécutée à l'aide de l'infrastructure. Les annotations présentes dans les classes compilées (i.e. le bytecode) sont interprétées lors de leur chargement en mémoire. L'utilisation des annotations Java permet ainsi de qualifier le code sans changer intrinsèquement son comportement, mais en permettant de le changer néanmoins lorsque cela est nécessaire.

### *III.2.3 Appel des méthodes du Manager : méthode "directe" de la programmation répartie*

La possibilité de définir le comportement réparti de l'application au travers des annotations rend plus facile la programmation d'applications distribuées, cependant, certains types de communications ont besoin de mécanismes plus étendus pour être mis en œuvre, en particulier l'utilisation d'objets distants dont le développeur ne connait pas l'existence ni le type lors du développement. Ainsi que des intergiciels le fournissent, comme CORBA avec son interface d'invocation dynamique (DII), nous permettons, au travers du modèle de programmation de notre infrastructure, au développeur d'utiliser plus dynamiquement l'intergiciel de communication mis à sa disposition. L'invocation dynamique de méthodes (i.e. l'appel de méthode inconnue lors du développement) est possible par le biais de méthodes directement accessibles au programmeur via l'objet Manager de l'infrastructure. Cet objet offre également toutes les méthodes pour recréer explicitement les mécanismes générés automatiquement à partir des annotations de répartition.

L'utilisation de l'une ou l'autre des possibilités d'utilisation du modèle de programmation n'est pas exclusive. Ainsi un objet proxy instancié automatiquement via des annotations peut être utilisé soit par méthodes implicites

annotées, soit par des appels explicites aux méthodes du Manager, et vice versa. Cette alternative autorise une plus grande dynamicité du développement réparti, car en effet la simplicité du modèle annoté a un prix, celui de la relative staticité du déploiement. Les annotations étant définies une fois pour toutes dans la classe, tous les objets instanciés de cette classe présentent potentiellement le même comportement réparti. Permettant une plus grande réactivité, les appels explicites sont donc un bon complément aux annotations.

### *III.2.4 Paradigmes de programmation et patrons de conception*

Le modèle de programmation de notre infrastructure logicielle multi-modèles présente les différents paradigmes de programmation de type RPC, ainsi que des patrons de conception (design patterns) permettant au développeur d'établir des communications d'autres types entre les composants répartis d'une application, comme nous l'avons présenté succinctement dans notre approche au chapitre II.2.1. Si les mécanismes de programmation RPC sont bien représentés dans l'état de l'art et bien intégrés au développement, nous proposons une approche plus intuitive afin d'en simplifier encore d'avantage l'utilisation. Les patrons de conception fournis avec le modèle de programmation, étant en contact direct avec le cœur de l'infrastructure, optimisent les communications correspondantes à leurs mécanismes, assurant ainsi un fonctionnement plus optimal que si le développeur avait créé lui-même ces mécanismes, des données sont par exemple échangées par défaut dans les communications, alors que certaines de ces données ne sont pas exploitées lorsqu'on utilise ces patrons de conception ; e.g. la référence du lien retour d'une invocation n'a pas lieu d'être pour un mécanisme de files de messages.

#### *Invocation à distance*

Comme nous l'avons dit un peu plus tôt, les modèles de base des langages de programmation sont, en grande majorité, basés sur des appels de méthodes. Il est donc logique et naturel de permettre d'utiliser un modèle de programmation proposant des appels de méthodes sur les objets distants.

**Au plus près du langage de programmation**
Les mécanismes d'invocation à distance des intergiciels actuels permettent, une fois les objets instanciés, d'utiliser indifféremment des objets locaux et distants. Notre approche est identique, mais apporte également des simplifications en ce qui concerne le nommage, le référencement et l'instanciation, comme nous le détaillons dans les sections III.5.2 et III.8.

La programmation utilisant des objets répartis avec l'infrastructure multi-modèles ne varie pas de la programmation classique. Un programme local ne doit ainsi subir aucune modification de code source (excepté l'ajout des annotations) pour devenir réparti.

Comme pour tout intergiciel RPC, la question de synchronisation et d'accès concurrents se pose. En effet, si dans un programme séquentiel local un objet n'est utilisé que par un fil d'exécution, il est susceptible en contexte réparti d'être invoqué simultanément (ou tout du moins parallèlement) par plusieurs objets appartenant à différents fils d'exécution.
Le langage Java prévoit une synchronisation explicite (au travers du qualificatif `synchronized`). Son utilisation restera identique par l'infrastructure.
Lorsqu'une synchronisation plus implicite est souhaitée, l'ajout de mécanismes ad hoc par le développeur, ou l'utilisation d'un système transactionnel est requis – ce qui est également le cas lors de l'utilisation classique d'un intergiciel actuel.

**Invocation synchrone bidirectionnelle**
Le modèle de programmation RPC fournit principalement un type d'invocation synchrone, similaire à une invocation locale. Ce type d'appel est bloquant jusqu'à ce que la méthode appelée ait terminé son exécution et retourné une valeur s'il y a lieu. En cas d'absence de valeur de retour (i.e. méthode *void*), la méthode appelée est néanmoins exécutée intégralement avant de rendre la main à la méthode appelante. Ce principe étant celui des langages procéduraux basiques, un simple appel de méthode dans l'infrastructure correspond à une invocation distante synchrone bidirectionnelle.

**Invocation asynchrone "oneway"**
Le second type d'invocation est unidirectionnel, i.e. l'appel de méthode est effectué, mais la méthode appelante reprend la main immédiatement et n'attend pas la valeur de retour. Bien que ce type d'invocation implique un certain nombre d'incertitudes (l'appel est-il bien arrivé à l'objet distant ? la méthode s'est-elle exécutée correctement ? En combien de temps ? etc...), elle offre également un certain nombre d'avantages comme la poursuite immédiate de l'exécution de la méthode appelante, l'absence de mécanismes de gestion des exceptions.

Lorsqu'un appel doit être effectué de façon unidirectionnelle, ceci peut être fait selon plusieurs méthodes :

- Soit l'objet annoté définit déjà cette méthode comme unidirectionnelle ; l'appel sera alors unidirectionnel pour toutes les méthodes appelantes, sauf spécification explicitement contraire de la part de l'appelant.
- Soit l'annotation d'instanciation de l'objet distant (cf annexe A) définit explicitement que toutes les méthodes accessibles de l'objet le seront par des appels unidirectionnels.
- Soit la méthode appelante est annotée. Cette annotation spécifie à l'infrastructure que les appels de méthodes distantes contenus dans cette méthode seront unidirectionnels. Cette technique demande un usage précis, car si plusieurs appels distants sont contenus dans ces instructions, ils seront tous unidirectionnels.

**Invocation asynchrone : le principe de "future" [Hal85]**

Le troisième et dernier type d'invocation est asynchrone, ce qui signifie que les deux méthodes, appelante et appelée, possèdent chacune leur propre fil d'exécution. La mise en œuvre d'un tel modèle demande certaines adaptations de la méthode d'invocation. En effet, si les modèles synchrones bidirectionnels et unidirectionnels sont somme toute assez proches, le modèle bidirectionnel asynchrone nécessite de gérer séparément l'invocation et sa valeur de retour. Pour cela, deux types de comportement sont offerts à la méthode appelante :

- Si la méthode ne spécifie aucun comportement particulier, notre infrastructure effectue un simple décalage de l'étape bloquante. En fait, l'invocation asynchrone vise à réduire le temps d'attente du résultat, en effectuant la requête au plus tôt, et en utilisant la valeur de retour au plus tard. Dans cette optique, les classes utilisant des objets distants via un modèle de programmation asynchrone sont automatiquement modifiées (cf section IV.2.3) à leur chargement, et l'invocation est divisée en deux étapes. Dans un premier temps, l'invocation asynchrone est effectuée, et un objet suivant le principe des *futures* est donné en valeur de retour. Lorsque la valeur de retour doit réellement être exploitée par le code de la méthode appelante, soit le retour de l'invocation a déjà eu lieu et la valeur est utilisée immédiatement, soit il n'a pas encore eu lieu, et la méthode l'attend en bloquant son exécution. Ce comportement permet donc de simplifier l'usage d'une invocation asynchrone, en reportant le blocage de l'exécution au moment où il sera vraiment nécessaire.
- La méthode peut également faire appel à un patron de conception fourni par l'infrastructure : l'`AsynchronousInvocationHandler`. Cet objet permet de gérer les différentes étapes d'une invocation asynchrone : la méthode charge le Handler d'effectuer l'invocation, et des méthodes

permettent de renseigner sur l'état d'avancement de l'invocation et de récupérer la valeur de retour, soit par scrutation (méthode poll), soit par récupération bloquante (méthode get), soit en fournissant un objet implémentant l'interface AsynchronousReceiver, dont la méthode callback `doReceive(Object value, AsynchronousHandler source)` est appelée lorsque la valeur est disponible.

Là où les intergiciels existants ne fournissent que des méthode explicites de gestion d'un appel asynchrone (e.g. la méthode `send_deferred()` de la DII de CORBA), nous proposons en complément à ce système une gestion basique du retard de la valeur de retour de l'invocation, complètement transparente pour le développeur car automatiquement intégrée aux instructions de la méthode appelante.

*Files de messages*

Pour permettre un grand panel de possibilités de types de programmation, et donc assurer la programmation la plus efficace possible, nous proposons également dans notre infrastructure des patrons de conception intégrant les fonctionnalités des modèles de programmation asynchrones des intergiciels actuels. Nous permettons ainsi aux développeurs d'utiliser des mécanismes orientés messages (MOM) et en particulier un patron de type "file de messages".

Les files de messages autorisent l'envoi ordonné et asynchrone de messages entre deux composants. Dans ce modèle de programmation, le composant producteur de messages procède toujours à l'envoi explicite (par une méthode dédiée), et le composant consommateur peut également procéder à la réception explicite, ou bien fournir une méthode callback pour être prévenu dès qu'un message arrive. Comme pour l'invocation asynchrone, l'infrastructure logicielle multi-modèles intègre un patron de conception nommé `MessageQueue`. Cet objet prend en charge les communications entre composants par file de messages, en assurant un système d'identification par identifiant et mot de passe (comme les intergiciels actuels basés sur les messages).
Une solution également proposée consiste à se substituer à l'objet `MessageQueue`. Pour ce faire, l'objet se substituant doit présenter l'annotation `@MessageQueuePattern`, et désigner par les annotations `@QueueConnectMethod` et `@QueueReceiveMethod` les méthodes qui devront être utilisées par l'infrastructure. Cette solution permet de rendre plus claire la programmation de ce paradigme, mais interdit aux autres composants d'utiliser cette file de messages en tant que consommateur. Elle ne convient donc pas à tous les cas

d'utilisation.
Pour consommer un message, le composant peut choisir entre scruter périodiquement l'arrivée de messages, ou bien effectuer un appel de méthode bloquant, qui attend qu'un message arrive. Un message consiste dans notre infrastructure en un objet sérialisable (i.e. qui peut être décrit entièrement en décrivant en série tous ses attributs, et récursivement jusqu'à obtenir des types élémentaires) transmis au consommateur.

*Sujets d'intérêt (topics)*

Les sujets d'intérêt sont également un modèle de programmation orienté messages. Ils permettent à des composants producteurs et consommateurs de se transmettre des messages (qui, comme pour les files de messages, sont des objets sérialisables). La différence avec un système de file de messages est la possibilité de connecter plusieurs consommateurs et plusieurs producteurs au même lien de transmission de messages.

Plusieurs cas de figure sont envisageables lors de l'utilisation d'un système de sujet d'intérêt :

- Dans le cas où plusieurs producteurs envoient des messages sur le même sujet d'intérêt (architecture dite "fan-in"), ceux-ci sont transmis dans l'ordre, sans qu'une distinction soit faite entre les messages des différents émetteurs.
- Si plusieurs consommateurs sont abonnés au même sujet d'intérêt (architecture dite "fan-out"), chacun recevra chaque message transmis. Ce système permet de faire de la diffusion ciblée (multicast) en évitant la redondance d'informations.
- Comme pour les files de messages, les consommateurs peuvent soit procéder à la réception explicite de messages par méthode dédiée, soit fournir une méthode callback, qui sera appelée dès qu'un message est produit sur le sujet d'intérêt.

Similairement au système de file de messages, des méthodes dédiées sont disponibles auprès du patron de conception `MessageTopic`. Les composants peuvent choisir entre ce fonctionnement actif, ou bien un fonctionnement passif basé sur une méthode callback appelée lors de la réception d'un message. Le consommateur peut également se substituer à l'objet `MessageTopic` grâce aux annotations `@MessageTopicPattern`, `@TopicConnectMethod` et `@TopicReceiveMethod`. Cependant, l'intérêt de ceci est assez limité, en considérant que l'utilisation généralement faite des sujets d'intérêt est la diffusion à plu-

sieurs consommateurs : ce système interdit une telle utilisation, étant donné que seul l'objet qui s'est substitué peut consommer les messages.

*Evénements*

Nous fournissons les outils nécessaires à un autre modèle de programmation orienté messages : les événements. Il s'agit en fait d'une évolution du principe de sujets d'intérêts, basée sur la même architecture où plusieurs composants peuvent produire et consommer des messages. Les événements ajoutent un concept de sens de production des messages. En effet, le système d'événements permet aux consommateurs de demander la production d'un message, on parle alors d'un modèle en "pull" (tirer), qui est ici l'alternative au modèle traditionnel en "push" (pousser).

Hormis ce concept supplémentaire, le système d'événements présente les mêmes caractéristiques que le système de sujets d'intérêts. L'infrastructure multi-modèles propose également un patron de conception pour ces événements (`EventService`, intégrant les méthodes de production et de consommation de messages ainsi que les méthodes d'identification. Le modèle de fonctionnement en "pull" n'est utilisable que dans le cas où un seul producteur fournit une méthode callback chargée de lui demander la production d'un message. Si les conditions ne sont pas réunies, le demande de production déclenchera une exception.

*Espaces partagés*

Les espaces partagés sont le dernier paradigme de programmation répartie implémenté dans le modèle générique de notre infrastructure. Pour permettre l'utilisation de ce paradigme, l'infrastructure fournit un patron de conception `SharedSpace`. Ce patron offre les méthodes dédiées pour l'identification par identifiant et mot de passe, la recherche, l'ajout, la récupération et la suppression d'objets.

Les mécanismes de recherche offerts par les intergiciels existants proposant des espaces partagés sont assez poussés : ils proposent des recherches par modèle, i.e. le développeur fournit un objet modèle à la méthode de récupération d'objet. Ce modèle définit alors le type d'objet recherché (sa classe), et les attributs initialisés du modèle sont autant de restrictions à apporter au choix de l'objet à récupérer.

Le plus performant pour fournir ce type de mécanismes est de réutiliser ceux d'un intergiciel actuel ; la fourniture d'un paradigme d'espaces partagés dans le modèle de programmation requiert donc la présence d'un modèle de communication implémentant ce type de communications. Les mécanismes fournis actuellement utilisent un système associatif clé/valeur basé sur des tables de hashage. Les possibilités offerts alors sont certes moindres, mais ceci n'est qu'une proposition de fonctionnalités pour illustrer notre approche.

## *III.3 Observateurs de contexte*

Les mécanismes d'adaptation au contexte prennent une part importante dans l'infrastructure logicielle multi-modèles. Pour permettre aux applications réparties de rester fiables et performantes en mobilité, il est nécessaire de concevoir ces mécanismes aussi efficacement que possible.

Les observateurs de contexte sont en fait des objets instanciés dans l'infrastructure multi-modèles, permettant de rendre compte de l'état du contexte. Chaque observateur possède son propre fil d'exécution (leur processus est indépendant), et fournit à son rythme les données au Manager de l'infrastructure. Les observations de contexte sont des couples clé/valeur (de type `java.util.Map.Entry<String,Object>` associant à un nom symbolique une valeur entière, décimale ou booléenne (e.g. `{"CORBA.canAccessToNaming", false}`).

Pour permettre son adaptation, l'infrastructure a besoin d'obtenir des informations de contexte claires et complètes. Il est important pour cela de bien définir ce qu'est le contexte pour cette infrastructure, puis nous présenterons successivement les trois catégories d'observateurs permettant l'adaptation des communications de l'infrastructure au contexte.

### *III.3.1 Notion de contexte*

Les recherches sur la définition du contexte ont été et sont encore nombreuses et variées, à tel point qu'il est nécessaire de bien cibler les besoins de notre infrastructure en la matière, afin de ne pas s'écarter de son optique première.

Le contexte servant dans notre cas à décider quels modèles de communication seront utilisés, ce sont ces mêmes communications qui nous donneront le plus de renseignements sur les capacités et limitations du réseau. Ce dernier

peut en effet présenter certaines caractéristiques qui rendent quelques modèles de communication moins performants, voire complètement inefficaces.

Bien que nous parlions du contexte, il s'avère que l'infrastructure sera souvent en présence de différents contextes simultanés. En effet, le contexte se résumant aux caractéristiques des liens de communication (donc du réseau), l'application présente certainement des composants répartis déployés dans des contextes distincts. Les observations de contexte sont donc à considérer relativement à chaque lien de communication, e.g. si la communication RMI est impossible avec un objet distant, cela ne signifie pas pour autant qu'aucun objet distant ne pourra être contacté via RMI, mais que cet objet en particulier devra être joint via un autre modèle.

### *III.3.2 Observations liées aux modèles de communication*

Dans un premier temps, la perception du contexte des communications entre les composants répartis se résume à l'observation de l'utilisation des modèles de communication. Cette observation consiste à noter les changements notables à plusieurs niveaux :

- Les différents modèles de communication dépendent en général d'un serveur central, qu'il ait une utilité directe de communication (e.g. le serveur JMS stocke les messages) ou bien qu'il assure un service (e.g. le serveur de nommage RMI, *rmiregistry*, permet de récupérer la référence d'un objet réparti à partir de son nom symbolique, ou bien en listant le contenu du serveur). Lorsque la communication avec ce serveur est impossible, il est relativement simple de penser que les autres communications via le modèle de communication concerné ne vont probablement pas aboutir (ce qui est évidemment le cas pour JMS, et probablement le cas pour RMI).
- Les modèles de communication présentent tous des systèmes d'erreurs et d'exceptions. Il est intéressant de savoir si ces erreurs surviennent, voire même à quel rythme, afin de prendre les mesures qui s'imposent en conséquence. Un modèle déclenchant souvent des exceptions se montre d'une part moins efficace, mais signifie d'autre part sûrement des problèmes réguliers d'établissement des communications.

Les observateurs liés aux modèles de communication sont un maillon important des mécanismes d'adaptation : ils effectuent un diagnostic en temps réel de la bonne "santé" des communications via ces modèles.

### *III.3.3 Observations liées au matériel et aux couches basses*

Le contexte d'exécution à prendre en compte dans le cadre de notre infrastructure se compose ensuite de l'état du réseau de communication. Cet état peut être défini selon plus ou moins de nuances, selon les nécessités et les conséquences de l'observation, et le matériel et les couches considérées.

Ainsi par une observation élémentaire, l'infrastructure peut savoir si le réseau est actif ou non. Cette observation semble être triviale, mais est cependant indispensable pour décider ou non d'utiliser un modèle de communication synchrone.

Pour fournir un état plus nuancé du réseau, une observation de la couche IP est en mesure de relever certaines statistiques permettant de chiffrer certains indices représentatifs des performances et de la fiabilité des communications à un instant donné :

- Le pourcentage d'erreurs CRC (ou checksum) recensées dans un laps de temps informe sur la qualité de la connexion. En effet, un taux important d'erreurs est généralement causé par une détérioration de l'intégrité des données échangées.
- Le temps de réception de l'acquittement (ACK) lors d'une communication : lorsqu'un paquet de données TCP/IP est envoyé, le serveur retourne un message élémentaire au client lui signifiant le succès de l'opération. Ce temps peut être considéré comme le temps que prend un aller retour de données (i.e. équivalent au "ping"). Plus ce temps est long, plus la communication synchrone va pénaliser le fil d'exécution du client et du serveur (temps de latence pour chaque envoi de données). L'utilisation d'un modèle de communication moins couplé peut être pertinent lorsque ce temps dépasse un certain seuil.
- En considérant toujours les acquittements TCP/IP, la prise en compte de l'absence de ces paquets ACK (timeout) peut présager une rupture de connexion avec le serveur. Par extension, une absence continue de ces paquets peut signifier une déconnexion permanente, alors que des absences peu fréquentes mais régulières peuvent signifier des déconnexions intempestives. Le choix du modèle de communication peut alors être différent en fonction des situations rencontrées.

Enfin, dans le cadre d'une connexion sans-fil de type Wi-Fi, WiMax ou HSDPA/UMTS, l'état des communications sur le réseau peut également être diagnostiqué via des données spécifiques fournies par les couches matérielles (pilotes de périphériques réseau), pour enrichir la perception du contexte par

l'infrastructure. Ainsi les pilotes de périphériques sans-fil fournissent deux données importantes, qui sont la qualité du signal (ou puissance du signal, souvent liée), et le débit atteignable sur le canal de communication.

#### *III.3.4 Observations définies par le développeur*

Les observations sont ici limitées à des données communes à tous les systèmes, ne prenant en compte ni le domaine d'application du programme, ni les possibilités techniques particulières que la plateforme d'exécution pourrait offrir. C'est pourquoi pour compléter ces observations "génériques", nous proposons dans notre infrastructure la possibilité pour le développeur de définir ses propres observateurs.

Des observations internes au fonctionnement de l'application peuvent être ajoutées, comme l'état des différents composants, ou bien des préférences utilisateur. En effet, comme il en est souvent question dans la littérature, les préférences utilisateurs sont une part non négligeable du contexte à prendre en compte dans le cadre de l'adaptation dynamique, et les mécanismes de prise en compte de ces préférences ne peuvent être implémentés qu'au cours du développement de l'application elle-même.

Des observations plus construites que celles fournies par l'infrastructure peuvent être calculées en fonction d'autres observations. Si l'ont veut par exemple prendre en compte l'évolution (augmentation, diminution) de la qualité du signal sans-fil plutôt que son chiffre exact, une observation consistant en la variation de cette qualité peut être calculée.

### *III.4 Politiques d'adaptation*

L'infrastructure logicielle multi-modèles est capable d'obtenir des observations du contexte d'exécution. Pour lui permettre de s'adapter à ce contexte, il est nécessaire d'établir des politiques d'adaptation : pour un contexte donné, certains modèles de communication devront être utilisés plutôt que d'autres. Nous allons présenter le moyen que nous proposons au sein de notre infrastructure pour définir ces politiques pour que leur mise en œuvre soit simple, compréhensible et efficace.

#### *III.4.1 Format des politiques*

Le format de définition des politiques est important. D'une part, il permet au développeur (ou au responsable du déploiement) de spécifier le plus pré-

cisément possible le comportement que l'infrastructure devra adopter dans différents contextes, dans un langage facile et complet. D'autre part, il permet à l'infrastructure de pouvoir interpréter ce langage efficacement, et de détecter lorsque ces politiques ont changé.

*Fichier XML : lisibilité et interopérabilité*

Les politiques d'adaptation sont définies dans un fichier XML. Ce format et ce support ont été choisis pour leur utilisation équivalente sur les différentes plateformes d'exécution. Ils permettent une utilisation construite et organisée des données, tout en étant directement compréhensible par l'homme ("human readable"). De plus, les principaux langages de programmation le supportent : e.g. .Net, Php, Java.

Utiliser un fichier comme support permet un transport et une configuration simples : le fichier est déployé en même temps que l'application, et peut être modifié à tout moment, que ce soit lors de l'exécution de cette application ou non, par tout programme ayant un droit d'accès à ce fichier. L'infrastructure multi-modèles, par l'intermédiaire de son Manager, observe l'état de ce fichier, afin d'immédiatement prendre en compte les nouvelles politiques dès qu'une modification a été détectée

*Association d'une cause et d'une conséquence*

Le format XML permet une hiérarchie des données, organisée en nœuds imbriqués. Une politique est définie en premier lieu par une cause, représentée par un nœud père. De cette cause peuvent dépendre plusieurs nœuds fils : soit une ou plusieurs conséquences (action à entreprendre), soit une ou plusieurs autres causes à prendre en compte, de qui pourront dépendre également d'autres nœuds fils, et ainsi de suite. Cette imbrication permet d'associer des causes (opérateur logique "ET") afin d'affiner les critères d'application de la politique.

**Les causes : observations**

La cause d'une politique d'adaptation est une observation issue d'un observateur de contexte. Comme nous l'avons défini plus haut en section III.3, une observation est transmise au sein de l'infrastructure comme un couple clé/valeur. La clé est une chaine de caractère décrivant ce que représente la valeur associée, comme c'est le cas pour les variables d'environnement Java (du type *chemin.de.l.objet.puis.fonction.de.la.valeur*). La valeur correspondant à cette clé peut être de différents types : entier, décimal, booléen.

Chaque type de valeur permet de sélectionner les valeurs cibles activant cette politique : un booléen peut être soit vrai, soit faux ; un entier ou un nombre décimal peut être soit égal à une valeur exacte, soit inférieur, soir supérieur à cette valeur. Par le principe d'association de causes pour une politique comme nous l'avons décrit plus haut, il est possible également d'établir un intervalle de valeurs pour lesquelles la politique est active.

Une cause particulière est présente dans le fichier de définition, nommée "Default" (cf figure III.4). Cette cause est vraie en toutes circonstances, mais n'est pas évaluée de la même façon que les autres (cf section III.4.2). Elle définit la configuration "de base" des modèles de communication, i.e. les modèles activés par défaut et leur ordre de priorités.

```
<Policy>
 <Default/>
 <Result type="Enable">
  <Model priority="1">RMI</Model>
  <Model priority="2">JMS</Model>
 </Result>
</Policy>
```

*Fig. III.4:* Exemple de politique d'adaptation par défaut définie dans le fichier XML.

**Les conséquences : activations, priorités et autres réactions**
La conséquence d'une politique d'adaptation peut être de plusieurs types : elle peut simplement consister en l'activation ou la désactivation d'un modèle de communication ; la combinaison de modèles est alors enrichie ou appauvrie d'un modèle à utiliser pour établir les communications.

Si plusieurs modèles de communication sont activés simultanément, pour lever toute ambigüité, le système prévoit également un système de 5 niveaux de priorités permettant de hiérarchiser les modèles entre eux. Car si la configuration de base définit un ordre de priorité initial, il se peut que selon le contexte cet ordre doive changer ; ce système de priorité autorise un changement dynamique de hiérarchie des modèles de communication.

Enfin, la conséquence d'une politique peut être une autre réaction ; l'infrastructure permet ainsi d'appeler une méthode particulière en réponse à un

changement de contexte. Pour des raisons de simplification de l'application de ce principe, la méthode concernée doit être statique et ne demander aucun paramètre. Si ce système est principalement proposé pour enrichir les capacités d'adaptation de l'infrastructure, il peut permettre au développeur d'établir des systèmes d'adaptation à de tout autres paramètres, ou de rendre son application consciente du contexte d'exécution.

### *III.4.2 Application déterministe des politiques*

Le système de définition des politiques d'adaptation est relativement simple. Cette simplicité est voulue dans un souci d'efficacité. En effet, plus les politiques sont complexes à définir clairement, et plus l'infrastructure devra prendre de temps pour les interpréter, or pour être réactive et efficace en cas de changement de contexte, cette interprétation doit être rapide et sans ambiguïté.

La politique particulière "Default" définit la politique par défaut de l'infrastructure, i.e. le comportement à adopter en l'absence de tout ordre contraire. C'est donc cette politique qui est prise en compte en premier lieu. Cependant, dès qu'une politique dont la cause est vraie redéfinit une activation ou une priorité d'un modèle, la politique par défaut concernant ce modèle est ignorée.

#### *Eviter les conflits pour optimiser l'interprétation*

Pour éviter tout problème d'interprétation de plusieurs politiques contradictoires, le système est bridé volontairement. Ainsi dès qu'une politique dont la cause est vérifiée définit l'activation d'un modèle, plus aucune redéfinition de cette activation n'est prise en compte. Il en est de même pour les priorités. L'ordre des politiques dans le fichier est donc très important : il décide de la priorité des politiques entre elles.

#### *Evolutions possibles du système de politiques*

Ce bridage du système d'évaluation des politiques peut néanmoins paraitre comme un frein à certaines situations complexes, où le contexte est définit par plusieurs paramètres contradictoires mais néanmoins primordiaux. On peut donc penser à des systèmes de négociation permettant de trouver une décision satisfaisant la majorité des politiques (cf systèmes multi-agents), mais ce type de systèmes présente un coût assez important en terme de temps de calcul. Si Qinna [TBO05] propose un système de précalcul des différentes

situations possibles, le problème reste entier quant à la réévaluation des politiques à l'exécution. Donc même si le système de politiques d'adaptation que nous proposons ne permet pas un établissement parfait d'adaptation à toutes les situations, il semble néanmoins être une solution rapide et performante répondant aux besoins de notre infrastructure multi-modèles.

## *III.5 Combinaison de modèles de communication : les personnalités protocolaires*

En complément du modèle de programmation générique, notre approche multi-modèles se base sur une combinaison de modèles de communication pour offrir des capacités d'adaptation à l'infrastructure logicielle. Cette combinaison permet deux types d'adaptation lors de l'exécution de l'application :

- L'adaptation au modèle de communication rencontré lors de l'utilisation d'un composant réparti. Grâce aux différents modèles de communication gérés simultanément par l'infrastructure logicielle, celle-ci est capable de communiquer via le modèle correspondant à chaque composant.
- L'adaptation au contexte lors de la communication entre deux infrastructures logicielles multi-modèles. Chaque infrastructure permettant d'utiliser différents modèles, le choix peut être fait du ou des modèles à utiliser en fonction du contexte, et ceci à chaque changement de contexte.

Pour gérer ces modèles de communication à partir du noyau neutre de l'infrastructure, il est nécessaire de mettre en œuvre différents mécanismes pour permettre une utilisation optimale par ce noyau (i.e. le Manager) des moyens de communication proposés par chaque modèle.

L'utilisation d'un modèle de communication implique la gestion des différents services proposés par ce modèle, comme la communication, mais aussi le système de références et de nommage. Le noyau neutre offre une abstraction de ces services pour qu'ils soient utilisables par le modèle de programmation, mais ces fonctions abstraites doivent correspondre le plus efficacement possible aux fonctions des modèles de communication.

Dans cette partie, nous allons parler d'objets et d'invocations. Bien que ces termes soient empruntés au champ lexical des modèles d'appel de procédure à distance (RPC), il peut s'appliquer tout aussi bien aux autres modèles.

Ainsi une invocation sur un objet "file de messages" désigne la production d'un message sur cette file.

### *III.5.1 Interception et redirection des communications*

Pour que les fonctions abstraites du noyau utilisent les communications offertes par les différents modèles, deux approches sont envisageables :

- Les modèles de communication répondent à des spécifications précises définissant un ensemble de services, qui ne sont pas tous ici nécessaires à l'utilisation que nous faisons de ces modèles. Un composant "communication" peut donc être implémenté pour répondre à chaque spécification de type de modèle de communication. Ce composant sera alors utilisé par les services abstraits du noyau pour permettre les communications de l'infrastructure avec le modèle concerné.
  Si cette approche permet d'implémenter un composant proposant seulement les fonctionnalités requises par l'infrastructure, et donc d'optimiser le fonctionnement global de celle-ci, elle demande un développement conséquent spécifique à chaque modèle. On trouve cette approche dans plusieurs projet comme Jonathan et PolyORB, mais ce n'est pas celle choisie pour notre infrastructure.
- Les intergiciels existants fournissent une implémentation complète répondant aux spécifications des services d'un modèle de communication. Le noyau neutre de notre infrastructure peut retranscrire les différentes communications du modèle de programmation au travers des méthodes fournies par les intergiciels.
  Cette seconde approche semble être moins optimale, cependant beaucoup d'intergiciels permettent l'accès à une couche intermédiaire entre le modèle de programmation et le modèle de communication. Cette couche peut prendre différentes formes : des intercepteurs, une interface d'invocation dynamique (DII) et son équivalent pour l'objet serveur (DSI).

Nous avons choisi d'opter pour cette dernière approche pour l'intégration des différents modèles de communication dans l'infrastructure. L'utilisation du service de communication d'un intergiciel est faite via la couche d'invocation intermédiaire fournie par celui-ci, permettant une intégration rapide des modèles voulus, et profitant ainsi de l'implémentation éprouvée d'un intergiciel déjà existant. Une invocation est envoyée par le service d'invocation dynamique lorsqu'il est disponible (ou par le système offert par le modèle de programmation classique de l'intergiciel le cas échéant), et reçue côté serveur

grâce à l'interface serveur dynamique (DSI) (ou par un objet proxy généré par l'infrastructure si aucune DSI n'est disponible).

Les exceptions générées par les invocations effectuées par les différents intergiciels sont gérées au plus tôt ; soit par interception des flux de données lorsque des intercepteurs sont fournis, soit par récupération d'exceptions directement au niveau de l'appel à l'intergiciel. Ces exceptions fournissent des renseignements relatifs à l'invocation elle-même, et si chaque modèle de communication génère ses exceptions spécifiques, elle peuvent être traduites en exceptions génériques (cf section III.7.3).

### *III.5.2 Enrichissement des références des objets : référencement combiné*

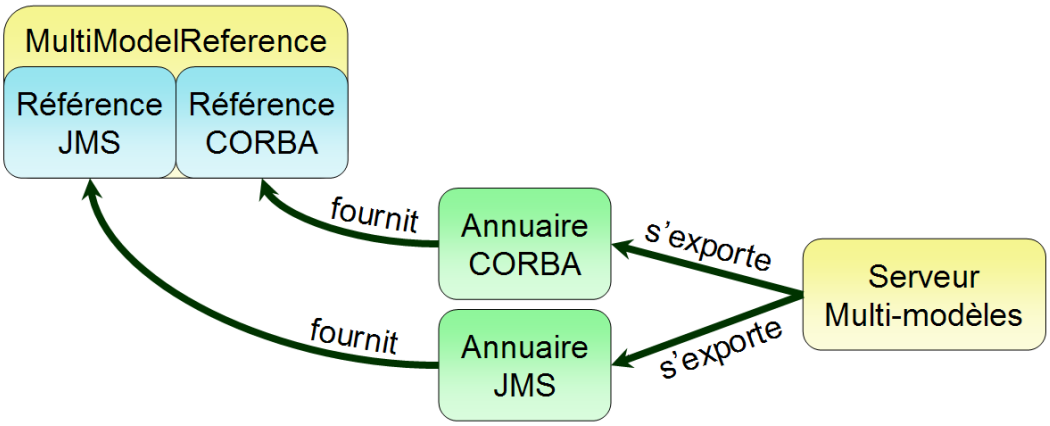


*Fig. III.5:* Schéma d'illustration de l'enrichissement des références multi-modèles.

Du point de vue d'un modèle de programmation, la référence d'un objet est un objet proxy (ou *souche*)permettant l'invocation à distance de ses méthodes. Le modèle de programmation générique que nous proposons présente une référence similaire, par le biais d'une annotation @ResolveObject, également sous la forme d'un objet proxy.

Cependant, les références peuvent prendre différentes formes, selon leur utilisation :

- Il peut s'agir d'un nom symbolique. Dans ce cas, la référence de l'objet est obtenue via l'utilisation d'un service d'annuaire, qui saura fournir l'objet proxy correspondant. Dans le cadre de notre infrastructure, le

nom symbolique est fourni aux services d'annuaire des différents intergiciels afin d'obtenir le ou les objets proxy correspondants (si l'objet distant est situé au sein d'une infrastructure multi-modèles, il peut être accessible via plusieurs modèles de communication, donc nécessiter plusieurs proxies). Ce (ou ces) proxy une fois obtenu, le modèle de programmation de l'infrastructure fournit un objet proxy unique, permettant aux invocations d'adopter un comportement multi-modèles.
- Il peut s'agir d'une référence textuelle. En général les modèles de communication fournissent une représentation "sérialisable" de objets distants sous la forme d'un URI par exemple (e.g. CORBA fournit un type de référence textuel nommé `corbaloc` issu du même principe). L'infrastructure multi-modèles fournit ce type de référence également : une concaténation des références textuelles des différents modèles de communication permet à l'infrastructure de recréer un proxy à partir de cette simple chaine de caractères.
- Enfin, la référence d'un objet distant peut prendre la forme d'un objet (e.g. ObjectReference) contenant toutes les informations utiles à la génération d'un proxy, mais utilisable seulement par l'intergiciel lui-même (le modèle de programmation ne donne pas accès à ces différentes informations à l'application). L'infrastructure utilise également ce type d'objet, appelé `MultiModelReference`. Ce fichier sérialisable contient les `ObjectReference` des différents modèles utilisés, ainsi que les différentes références textuelles, et les informations relatives aux propriétés multi-modèles de l'objet distant.

Un problème important se pose lorsqu'un seul service d'annuaire est capable de fournir l'objet proxy correspondant à l'objet distant. En effet, cet objet peut, ou non, être capable de communications multi-modèles. Pour le savoir, et connaître la référence de cet objet pour les autres modèles disponibles, l'infrastructure devra attendre de s'y connecter, afin de pouvoir dialoguer avec son Manager (cf section III.6.4).

## *III.6 Le Manager*

Comme nous l'avons décrite, l'infrastructure multi-modèles s'articule autour d'un noyau neutre, exempt de toute personnalité applicative ou protocolaire, chargé de mettre en relation le modèle de programmation générique et la combinaison de modèles de communication. Ce noyau neutre est représenté par le composant *Manager*. Ce Manager se charge de plusieurs rôles dans l'infrastructure, tantôt courtier responsable de l'acheminement des in-

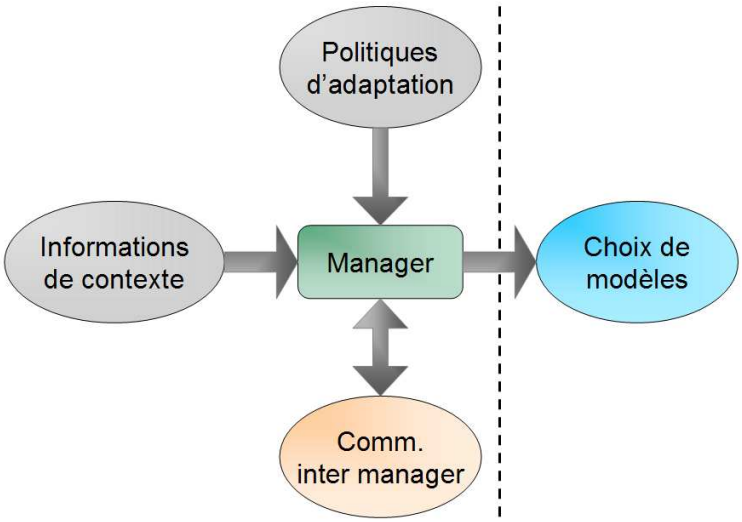


*Fig. III.6:* Rôle central du manager dans l'infrastructure.

vocations, tantôt gestionnaire d'adaptation mettant en application les politiques d'adaptation, et tantôt représentant distant de l'infrastructure pour les autres composants répartis.

Dans cette section, pour simplifier les explications, nous partons du principe que l'infrastructure est chargée de gérer des invocations partant d'une source, le client, et à destination d'un serveur. Les modèles de programmation répartie autres que RPC s'appliquent également à ce modèle par extension, ainsi dans une file de message, le producteur peut être considéré comme le client, et le consommateur comme le serveur.

### *III.6.1 Un rôle central dans l'infrastructure : Courtier*

Le Manager occupe principalement le rôle de courtier (généralement désigné par *broker* dans la littérature, terme anglais équivalent) au sein de l'infrastructure logicielle multi-modèles. Il est chargé d'acheminer les invocations du modèle de programmation (client) au modèle de communication, et du modèle de communication au modèle de programmation (serveur). Dans cette tâche, il est aidé par les classes proxy client (définies par l'annotation @ResolveObject) et les classes proxy serveur (annotation @DistributedObject). En effet, les différents types d'invocation ainsi que les mécanismes particuliers des patrons de conception sont transmis au courtier par le biais d'une invocation interne (cf section III.7), qui uniformise les mécanismes de l'invocation indépendamment du type d'invocation source.

Côté serveur, le Manager gère également la transmission de l'invocation. Une fois reçue par un ou plusieurs modèles de communication, l'invocation est transmise au Manager qui génère une invocation interne correspondante, transmise alors au proxy serveur de l'objet concerné afin qu'il effectue l'invocation comme convenue par le modèle de programmation.
Le retour de l'invocation suit le chemin inverse : la valeur de retour est ajoutée à l'invocation interne puis celle-ci est renvoyée au client via le Manager du serveur, le ou les modèles de communication, le Manager du client, et le proxy client.

Le Manager doit également jouer le rôle de courtier pour les références et le service de nommage. Dans un premier temps, comme nous l'avons expliqué précédemment en section III.5.2, le Manager fournit au modèle de programmation un objet `MultiModelReference` nécessaire à l'infrastructure pour générer l'objet proxy pour le client. Basé sur les références fournies par les modèles de communication, cet objet permet à l'objet proxy de fournir au Manager le moyen, via chaque modèle de communication disponible, d'accéder à l'objet distant. Le Manager agit de la même façon pour le service de nommage : lorsqu'un nom symbolique est fourni au modèle de programmation, le Manager effectue la requête sur les services de nommage des modèles de communication, et génère un objet `MultiModelReference` composé des références retournées par ces services. Lorsqu'un serveur indique sous quel nom symbolique il veut s'enregistrer grâce à l'annotation @DistributedObject, le Manager effectue l'action d'enregistrement ("binding") auprès des services de nommage pour que ce nom se réfère à l'objet pour tous les modèles de communication utilisés.

### *III.6.2 Choix de(s) modèle(s) de communication : Gestionnaire d'adaptation*

Le rôle novateur de ce Manager est celui de gestionnaire d'adaptation. En effet, en addition des mécanismes de communication entre le modèle de programmation et les modèles de communication, ce noyau de l'infrastructure est en charge des mécanismes d'adaptation au contexte. Pour cela, le Manager reçoit les observations de contexte de la part des observateurs par le biais d'une méthode `submitObservation(java.util.Map.Entry<String,Object>)`. En effet, comme nous l'avons spécifié dans la section III.3, chaque observateur de contexte possède son propre fil d'exécution ("thread"), et soumet à son rythme ses observations via cette méthode du Manager.

Une fois l'observation reçue, le nouveau contexte[1] mis à jour est soumis aux politiques d'adaptation. Ces politiques, définies par l'utilisateur sous la forme d'un fichier XML, sont traduites sous forme d'un ensemble ordonné d'objets `AdaptationPolicy` permettant d'obtenir un objet résultant `ContextAdaptation` qui définit les modèles de communication activés, ainsi que la priorité de chacun.

Le Manager redéfinit donc la hiérarchie des modèles de communication en fonction des observations de contexte, au fur et à mesure de leur réception. De plus, si le fichier de politiques est modifié (soit par l'application elle-même, soit extérieurement), un nouvel ensemble ordonné de `AdaptationPolicy` est construit, et donc un nouvel `ContextAdaptation`. Le Manager joue donc un rôle de gestionnaire d'adaptation réactif au contexte.

### *III.6.3 Communication multi-modèles : le changement de modèle en cours d'invocation*

Dans le cas où l'application et le composant distant utilisent une infrastructure logicielle multi-modèles, chacun présente donc un ensemble de modèles de communication. Selon les politiques d'adaptation et le contexte, un ou plusieurs modèles sont susceptibles d'être utilisés. Si plusieurs modèles sont choisis, cela permet au Manager de pouvoir faire appel à un autre modèle en cas de défaillance du premier.

Comme nous le définissons dans la section III.7, l'invocation interne fournit entre autres informations la méthode de réponse via les différents modèles de communication utilisés (e.g. retour RPC avec le numéro identifiant de la requête, file de messages de réponse). Ceci permet aux Managers de gérer des changements de modèles en cours d'invocation. Ainsi deux cas peuvent se présenter :

- Soit un échec (exception, erreur) intervient à l'étape de l'envoi de l'invocation de la part du modèle de communication. Dans ce cas, le Manager client opte pour le modèle suivant dans l'ordre défini par les politiques d'adaptation, et soumet à nouveau l'invocation.
- Soit un échec (mêmes cas) intervient lors de l'envoi de la réponse à l'invocation. Le Manager serveur opte alors pour le modèle suivant parmi l'intersection des modèles activés selon ses politiques et des modèles dont la méthode de réponse a été spécifiée dans l'invocation interne.

[1] Le contexte est donc ici l'ensemble des observations soumises par les observateurs de contexte.

Ce rôle supplémentaire du Manager offre une fiabilité supplémentaire par rapport aux modèles de communication traditionnels, et permet ainsi d'éviter les erreurs dues aux changements de contexte n'ayant pas pu être observés et traités à temps par l'infrastructure.

#### *III.6.4 Communication inter-manager*

Pour permettre cette adaptation au contexte même au cours d'une invocation, il est nécessaire pour les Managers de communiquer entre eux. En effet, l'adaptation ne peut être efficace que si elle est homogène d'un bout à l'autre d'un lien de communication, et cette communication inter-manager permet à plusieurs infrastructures d'adopter le même comportement.

Cette communication entre les noyaux de plusieurs infrastructures consiste essentiellement en un échange de la liste des modèles de communications activés ou non, permettant ainsi d'établir un sous-ensemble commun de modèles utilisables pour communiquer entre ces infrastructures.

Par cette communication répartie, des observateurs peuvent également soumettre des observations aux Managers distants ; observations qui peuvent ainsi être prises en compte par les politiques d'adaptation, pour définir plus précisément les comportements à adopter grâce à une connaissance globale du contexte des communications.

Ces communications se matérialisent sous plusieurs formes :

- D'une part, le contenu des objets `MultiModelInvocation` (cf section suivante, III.7) permet aux Managers d'échanger l'état des invocations en cours, ainsi que les modèles de communication disponibles pour communiquer (via l'objet `MultiModelReference`).
- D'autre part, chaque Manager peut utiliser la combinaison de modèles de communication lui-même pour effectuer des invocations sur d'autres Managers (i.e. chaque Manager est lui-même accessible via les modèles de communication). Chacun propose ainsi des méthodes pouvant être appelée par d'autres Managers afin d'envoyer ou de recevoir des informations.

## *III.7 Les communications internes à l'infrastructure : l'objet MultiModelInvocation*

Les invocations apparaissent dans le modèle de programmation comme de simples appels de méthodes, l'infrastructure logicielle multi-modèles a néanmoins besoin de transporter un bon nombre d'informations avec chaque invocation afin de permettre une exécution et une exploitation performantes de celle-ci.

Nous allons tout d'abord exposer les raisons de ce besoin d'ajout d'informations à l'invocation, puis nous décrirons l'objet permettant cette invocation au sein de l'infrastructure multi-modèles, et enfin nous nous attarderons sur un attribut particulier de cet objet, clé de voûte de notre approche multi-modèles.

### *III.7.1 Nécessité d'un protocole de communication interne*

Lors d'une invocation distante un intergiciel a besoin d'ajouter un minimum de données liées à cette invocation, ne seraient-ce qu'un identifiant (indice) de l'invocation, ainsi que la référence de l'objet appelant, et de l'objet appelé. Dans le cadre de l'infrastructure multi-modèles, de telles données sont également indispensables, afin de situer cette invocation.

Ces données doivent être transmises du client au serveur et vice-versa, mais doivent également transiter au sein même de l'infrastructure. En effet, plusieurs composants gèrent tour à tour l'invocation, et ceux-ci doivent transmettre des données annexes afin d'assurer une bonne exécution à cette invocation. C'est pourquoi nous proposons le protocole de communication interne relatif aux invocations comme décrit dans ce qui suit.

### *III.7.2 L'objet MultiModelInvocation, une mine d'informations enrichie tout au long de la communication*

Pour transmettre une invocation au sein de l'infrastructure, un objet MultiModelInvocation permet de faire transiter toutes les informations nécessaires à la communication et l'exécution d'une invocation.

Cet objet est tout d'abord sérialisable, i.e. tous ses attributs peuvent être décrits l'un après l'autre afin d'être enregistrés (persistance) ou transmis

(communication). Relativement à l'infrastructure, cela permet au MultiModelInvocation d'être envoyé à l'infrastructure serveur dans le cas d'une communication multi-modèles.

L'objet ainsi proposé permet tout d'abord de décrire l'invocation par la désignation de l'objet appelant (références complètes de l'objet, y compris sa classe), ainsi que l'objet et la méthode appelée (références complètes également, sous la forme d'une `MultiModelReference` et d'un nom absolu de méthode, e.g. `public java.lang.Object.toString()java.lang.String ;`). La comparaison des paquetages de l'objet appelant et appelé permet d'assurer un fonctionnement fidèle aux règles d'accès définies par Java. Ainsi, si ces paquetages sont incompatibles et que la méthode appelée est de type `protected`, une exception est déclenchée afin de notifier la violation.
Les paramètres de l'invocation sont également spécifiés, sous la forme d'un vecteur d'objets. Ces objets doivent être soit sérialisables, soit objets répartis via l'infrastructure.
Un identifiant unique spécifique à cette invocation est aussi présent, afin de pouvoir l'identifier tout au long de son parcours (et éviter les erreurs d'invocations multiples).

Des données sont ajoutées à cette invocation au fur et à mesure de son cheminement dans l'infrastructure. Ainsi, le modèle de programmation ajoute le type d'invocation : synchrone, asynchrone, one-way, future, patron de conception. Le manager ajoute en conséquence (ou non, selon le type d'invocation) la ou les références pour envoyer la valeur de retour de l'invocation.

S'il s'agit d'une communication classique (un seul modèle de communication), l'utilisation de l'objet MultiModelInvocation s'arrête au modèle de communication, qui transmet alors une invocation classique. Par contre, s'il s'agit d'une communication multi-modèles, cet objet est transmis tel quel à l'infrastructure distante, qui peut alors ajouter la valeur de retour à l'invocation (et supprimer les paramètres, pour réduire la quantité de données à transmettre) pour la retourner : soit par le modèle de communication initial (e.g. retour CORBA), soit par l'une des références de retour spécifiées dans l'objet.

### *III.7.3 L'attribut ReturnValue*

Lorsqu'il s'agit d'une communication multi-modèles, l'objet `MultiModelInvocation` est transmis à l'infrastructure serveur. Celle-ci transmet l'invocation à l'objet proxy serveur, qui l'effectue comme l'objet serveur l'a spécifié

(e.g. soit RPC, soit événement). S'il s'agit d'un appel unidirectionnel "one-way", l'invocation est terminée. Dans les cas où une valeur de retour est nécessaire, l'objet proxy serveur utilise l'attribut `Object ReturnValue` pour la transmettre à l'appelant. Si l'invocation a réussi, cet attribut contient cette valeur de retour (même en cas d'une fonction `void`, l'objet `MultiModelInvocation` est retourné avec `ReturnValue` à `null`). Si une exception a été déclenchée, celle-ci est alors placée dans `ReturnValue`.

L'itinéraire retour de `MultiModelInvocation` prend en priorité le même chemin qu'à l'aller : s'il a été transmis via le modèle CORBA, il est renvoyé dans le même fil d'exécution (la même invocation CORBA). Les intercepteurs du modèle informent le Manager de la réussite ou non de l'envoi de la réponse. Si ce n'est pas le cas, celle-ci est envoyée via le premier modèle activé de la combinaison de modèles de l'infrastructure, qui soit également défini par les références retour du `MultiModelInvocation`, et qui ne soit pas CORBA.

Lorsque `ReturnValue` désigne la valeur de retour, celle-ci est alors soumise à l'appelant une fois le `MultiModelInvocation` reçu. Lorsqu'il s'agit d'une exception, deux cas peuvent se présenter :

- L'exception a été déclenchée par l'usage normal de la méthode invoquée (e.g. OutOfBoundsException lors d'un appel sur une liste). Cette exception est alors soumise à l'appelant telle quelle, comme s'il s'agissait d'une exécution locale.
- L'exception a été déclenchée par le ou les modèles de communication utilisés, ou par l'infrastructure elle-même. Plusieurs cas sont possibles et sont alors regroupés dans un petit nombre d'exceptions clés. L'exception `NotMultiModelCommunicationException` est déclenchée lorsque l'objet `MultiModelReference` faisait état d'un serveur multi-modèle alors que ce n'est pas le cas. `NoSuchRemoteTargetException` est déclenchée si le serveur non multi-modèles n'est pas joignable par la référence proposée, ou si aucun des modèles proposés pour un serveur multi-modèles n'est utilisable. `NoSuchRemoteObjectException` correspond à un serveur joignable, mais aucun objet correspondant à la cible n'est présent sur le serveur.

## *III.8 A propos des cycles de vie : l'instanciation*

Les objets répartis, ainsi que les patrons de conception présentent plusieurs états, et nécessitent l'instanciation de plusieurs objets distincts pour les utiliser en répartition physique. En effet, une communication RPC demande,

en plus de l'instanciation de l'objet proprement dit, l'utilisation d'un objet proxy au niveau du client et du serveur. Pour les mécanismes de files de messages, une file doit être créée au niveau du serveur, et chaque producteur et consommateur utilise lui-même un objet proxy. Ces différents proxies doivent être gérés proprement, ainsi que le cycle de vie proprement dit de l'objet réparti.

### *III.8.1 Instanciation et libération du proxy*

Chaque objet proxy doit être instancié lorsque le modèle de programmation en a besoin, i.e. lorsque l'objet appelant définit sa connexion à l'objet appelé pour le proxy client, et lorsque l'objet appelé est défini comme étant réparti pour le proxy serveur.
Le proxy client peut être instancié de deux façons : soit en étant attribut annoté (@ResolveObject) d'un objet de l'infrastructure multi-modèles, dans ce cas il sera libéré lorsque l'objet possesseur sera libéré ; soit instancié explicitement à l'aide de l'usine ("factory") fournie par le Manager afin d'assurer l'utilisation à la volée d'objets répartis. Dans ce cas, le proxy sera libéré lorsqu'il ne sera plus référencé par le modèle de programmation, ou bien explicitement par la méthode `releaseProxy(Object p)` du Manager.

Le proxy serveur peut être lui aussi instancié de deux façons similaires : soit en étant un attribut annoté @DistributedObject, et sera alors libéré lorsque son possesseur le sera également, soit en étant instancié via une méthode `provideObject` dédiée du Manager, et sa libération interviendra lorsqu'il ne sera plus référencé et qu'il aura invoqué la méthode `unprovideObject`.

### *III.8.2 Cycle de vie de l'objet réparti*

Le cycle de vie d'un objet réparti dans un intergiciel est le temps durant lequel l'objet est accessible à distance, i.e. l'intergiciel est capable d'établir des communications avec celui-ci. Les méthodes `provideObject` et `unprovideObject` du modèle de programmation permettent de spécifier quand l'objet doit être accessible. Ces méthodes assurent l'accessibilité au travers des différents modèles de communication utilisés. Ces méthodes explicites permettent au développeur de gérer clairement le cycle de vie des objets répartis au sein de l'infrastructure.

## *III.9 Exemples et illustrations du principe : les intergiciels existants*

Comme le métamodèle de notre infrastructure le présente, ses mécanismes de fonctionnement permettent d'endosser plusieurs personnalités applicatives et protocolaires, i.e. plusieurs modèles de programmation et de communication. La composition de cette infrastructure dépend d'une part du ou des modèles de programmation utilisés, et d'autre part des modèles de communication choisis via les politiques d'adaptation. Il en résulte alors une combinaison de modèle(s) de programmation et de communication conforme aux besoins et affinités des développeurs, et adaptée au contexte de déploiement de l'application.

Dans certains cas particuliers d'utilisation de notre infrastructure logicielle multi-modèles, on peut ainsi obtenir un intergiciel en tous points comparable à un intergiciel actuel. Nous allons donc présenter deux intergiciels principalement utilisés aujourd'hui, "recréés" à partir de notre infrastructure, pour illustrer le principe de généricité de notre approche, ainsi qu'un intergiciel multi-modèles présenté précédemment.

### *III.9.1 RMI : le "tout synchrone"*

Le modèle de programmation de référence pour les intergiciels de communication est sans nul doute le modèle RPC. RMI [Mic03] est un intergiciel synchrone offrant un modèle de programmation RPC, comme présenté en section I.2.3.
Le modèle de programmation générique de notre infrastructure permet aisément au développeur de choisir le ou les modèles à utiliser. Le modèle RPC est naturellement mis en œuvre grâce aux annotations, et devient totalement transparent une fois l'objet proxy instancié (cf section III.2.4). L'invocation au niveau client et serveur du modèle de l'infrastructure est donc bidirectionnelle synchrone.
Le modèle de communication utilisé par RMI peut être soit JRMP, soit IIOP (cf section I.2.3). La combinaison de modèles de communication de notre infrastructure, dirigée par les politiques d'adaptation, permet d'utiliser un ou plusieurs modèles de communication. L'utilisation des deux modèles proposés par RMI[1] se fait ici simplement en spécifiant RMI dans les politiques pour assurer les communications.

[1] Les détails de cette implantation sont donnés en section IV.3.1.

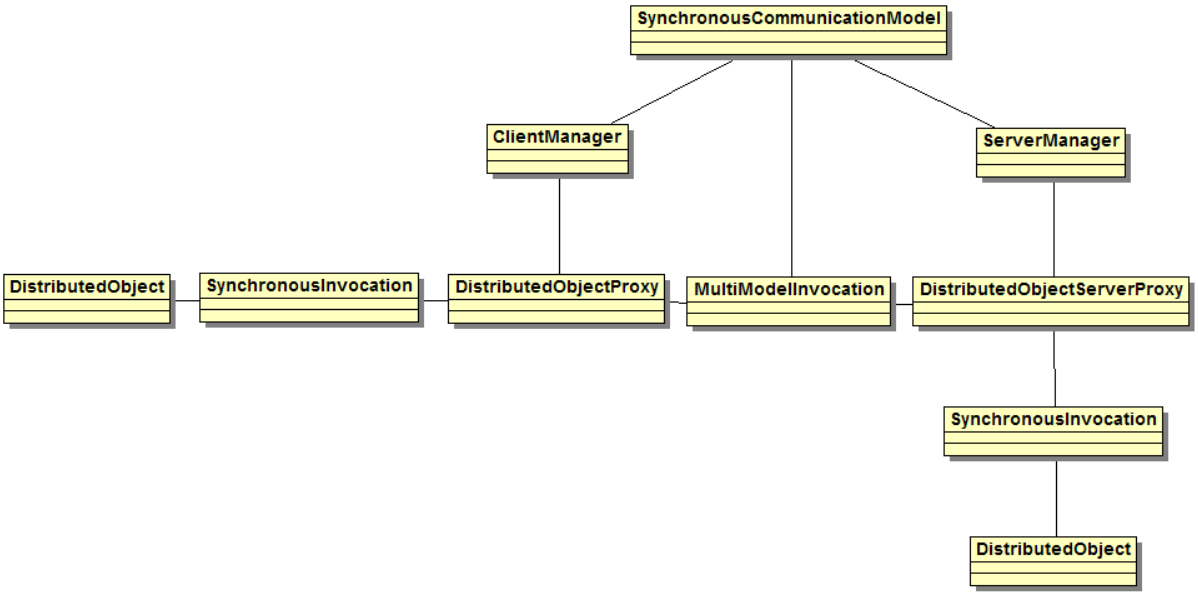


*Fig. III.7:* Configuration de l'infrastructure multi-modèles pour recréer un intergiciel de type RMI.

Le modèle d'infrastructure résultant de cette "émulation" de l'intergiciel RMI est donc entièrement synchrone. A l'instar de ReMMoC [GBS03], d'autres intergiciels RPC présentent une certaine dynamicité quant aux différents moyens de nommage, de découverte et de communication. Notre approche nous permet également d'offrir cette dynamicité à ce "pseudo RMI", il suffit pour cela d'activer d'autres modèles de communication dans les politiques, tout en plaçant RMI en tête des priorités. L'intergiciel obtenu est alors capable de se substituer à RMI, tout en offrant la possibilité d'utiliser des composants communiquant via d'autres modèles.

### *III.9.2 JMS : le "tout asynchrone"*

Un MOM[1] fournit un ou plusieurs types de patrons de conception d'envoi/réception de messages. JMS permet la communication via des files de messages (cf section I.4.3) et des sujets d'intérêt (cf section I.4.5). Ces deux patrons sont fournis par notre modèle de programmation générique, et dans l'optique de recréer un intergiciel tel que JMS, l'invocation au niveau client et serveur du modèle de l'infrastructure est donc uniquement unidirectionnelle

[1] *Message Oriented Middleware*, Intergiciel orienté messages.

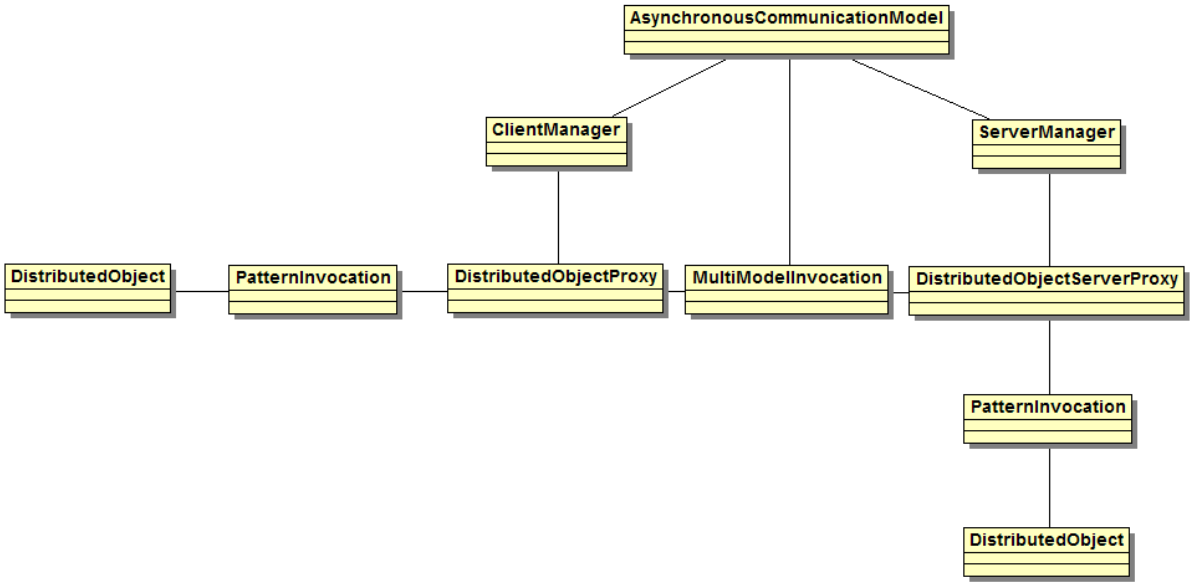


*Fig. III.8:* Configuration de l'infrastructure multi-modèles pour recréer un intergiciel de type JMS.

asynchrone.
Tout comme avec JMS, le développeur aura le choix entre une utilisation passive du système (par le biais d'une méthode "callback" qu'il fournit) ou active (en instanciant un objet proxy auprès duquel il produira et consommera les messages).

La politique d'adaptation permettant de calquer cet intergiciel déclare juste JMS comme seul et unique modèle de communication inconditionnel. Bien que cette politique n'autorise ici aucune faculté d'adaptation dynamique, une telle définition statique permet néanmoins de définir la personnalité protocolaire au lancement de l'application, comme le propose également PolyORB. Le changement de politique pendant l'exécution permet même de modifier ce choix au cours de l'exécution de l'application.

### *III.9.3 La commutation Synchrone / Asynchrone*

La proposition de commutation synchrone / asynchrone présentée en section I.3.8 part d'un système composé de deux paradigmes indépendants, et adapte les communications en cas de changement de contexte. En effet, sans intervention du système, les messages sont envoyés via un modèle de com-

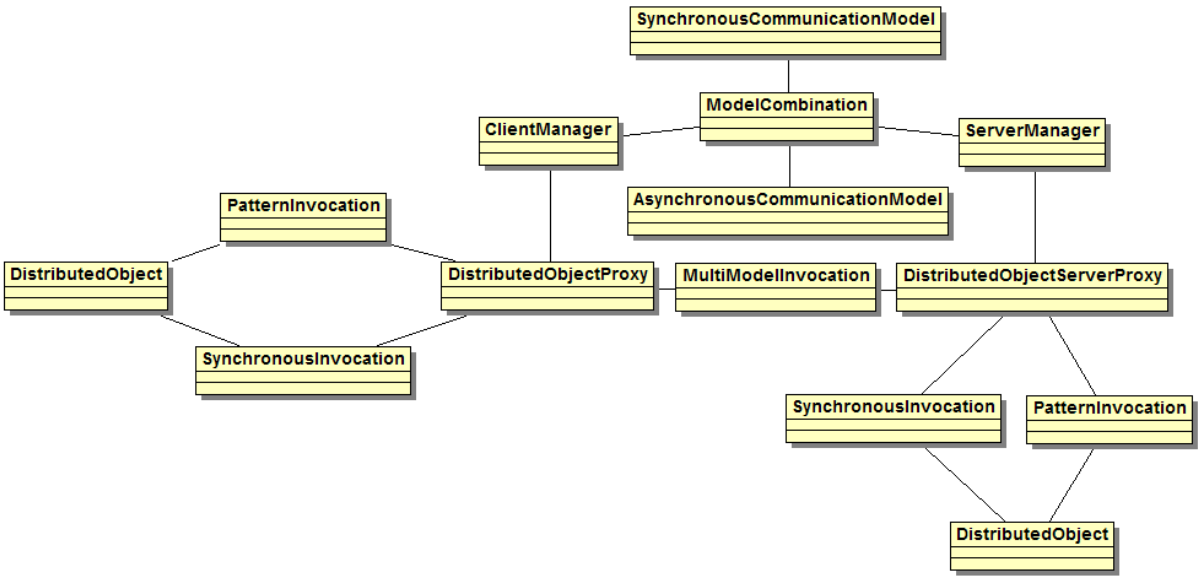


*Fig. III.9:* Configuration de l'infrastructure multi-modèles pour recréer une commutation synchrone / asynchrone.

munication asynchrone, tandis que les invocations bidirectionnelles sont effectuées via un modèle synchrone. Notre infrastructure, intégrant les deux modèles de programmation au sein d'un seul modèle générique, ne fait aucune distinction lors de l'utilisation du modèle de communication, ceci permet de profiter des performances du modèle synchrone lorsque celui-ci est disponible, et ce même pour un envoi de message. Les modèles de programmation synchrone et asynchrone sont ici tous deux présents dans le modèle de l'infrastructure.

Deux modèles de communication sont activés afin de permettre des communications synchrones et asynchrones comme le commutateur proposé, et le choix se fait en fonction du contexte, ici observé par le biais d'un système de ping/pong (les détails de ce mécanisme sont donnés en section IV.1.3).

Notre infrastructure multi-modèles permet donc de fournir un système de commutation synchrone / asynchrone comme le proposent V. Budau et G. Bernard, en autorisant de plus à changer de modèle de communication entre l'envoi d'une invocation et la réception de son résultat. L'intérêt supplémentaire de notre proposition est de pouvoir communiquer également avec

des composants utilisant seulement l'un ou l'autre des modèles utilisés. Cet aspect de compatibilité avec les composants ne présentant pas de fonctionnement multi-modèles est un bénéfice certain dans le développement et le déploiement d'une application distribuée hétérogène.

## *III.10 Synthèse de notre proposition d'infrastructure logicielle multi-modèles*

Pour appliquer notre approche multi-modèles, nous proposons une infrastructure logicielle présentant tous les aspects de celle-ci. Tout d'abord, son modèle de programmation générique basé sur des annotations et des patrons de conception autorise le développeur à choisir le ou les types de communications qu'il souhaite utiliser pour permettre à son application de communiquer.

Ensuite, son modèle de communication est composé d'une combinaison de modèles existants : chacun permet d'une part de se connecter aux applications l'utilisant (i.e. ayant été développées avec un intergiciel utilisant le modèle concerné) ; la combinaison de plusieurs modèles permet d'autre part d'assurer des communications performantes et fiables entre applications utilisant notre infrastructure. Cette combinaison dynamique permet de s'adapter en temps réel aux conditions, et d'ainsi opter pour le modèle le plus pertinent, qu'il soit ou non le même lors de l'émission et de la réception.

Des observateurs de contexte permettent de rapporter sous forme de données (e.g. valeurs numériques, états booléens) des propriétés du contexte d'exécution de l'application. Des politiques d'adaptation prennent en compte ces observations pour définir le ou les modèles les plus à-même d'assurer de bonnes communications dans ce contexte.

Elément central de notre infrastructure, le Manager intègre des mécanismes d'adaptation dynamique, et assure le respect des politiques d'adaptation. Il gère également l'acheminement des communications du modèle de programmation via la combinaison de modèles de communication. Ces communications sont assurées au travers d'un objet MultiModelInvocation contenant toutes les informations nécessaires à un traitement efficace de l'invocation.

Enfin, des exemples d'intergiciels "recréés" à partir de notre infrastructure complètent cette présentation en illustrant la capacité de l'infrastructure logicielle multi-modèles d'être "au moins équivalente" aux intergiciels actuels, tout en offrant des mécanismes supplémentaires.

# IV

# Travaux d'application et implémentations

Dans ce chapitre, nous présentons les différents travaux d'implémentation que nous avons effectués pour illustrer et justifier notre approche. En effet les fondements théoriques de notre architecture sont basés sur des projets existants et les notions proposées ont pour une grande part été mises en œuvre dans des systèmes libres ou commerciaux, cependant l'association de ces différentes notions n'a pour le moment jamais été implémentée.
La viabilité d'un système comme celui que nous proposons, ainsi que les performances qu'il peut offrir doivent par conséquent être démontrées. C'est pourquoi nous avons implémenté notre approche multi-modèles dans plusieurs applications.

Nous allons dans un premier temps présenter la première implantation de notre approche dans une application concrète : le greffon multi-modèles. Développé dans le cadre du projet Train-IPSat, ce greffon se propose de résoudre un problème d'adaptation au contexte de la mobilité dans les transports ferroviaires. Ce greffon ayant fait l'objet de tests de performances comparatifs, le bilan de ces tests permettent de chiffrer, et ainsi de juger de l'apport et du coût d'une approche multi-modèles par rapport à un système classique.

Nous présenterons dans un deuxième temps l'implémentation de l'infrastructure logicielle multi-modèles. Cette implémentation s'est déroulée en plusieurs étapes, correspondantes aux différentes briques composant l'infrastructure ; nous les expliquerons en détails, ainsi que les problèmes rencontrés

et les solutions adoptées.
Une fois l'infrastructure formée, des modèles de communication ont été implantés dans cette dernière, i.e. les mécanismes permettant de gérer efficacement ces modèles ont été développés. Nous exposerons les tenants et aboutissants de ces implantations de modèles synchrone (RMI) et asynchrone (JMS).

L'infrastructure logicielle multi-modèles a été implémentée en deux versions. Une première version propose une adaptation "précalculée" au contexte grâce à une combinaison "statique" des modèles de communication. Cette version comporte un intérêt principal qui est la prédictibilité.
Une deuxième version présente l'infrastructure multi-modèles dans sa version dynamique : les mécanismes d'adaptation sont générés au chargement de l'application, et les politiques sont prises en compte en temps réél. Nous exposerons donc ces deux versions, avec les spécificités techniques qu'elles nécessitent.

## *IV.1 Greffon multi-modèles*

Afin de valider notre approche multi-modèles, et d'en illustrer le principe dans une application concrète, nous avons conçu et développé un greffon multi-modèles destiné aux applications existantes, et en particulier à un proxy Internet embarqué à bord des trains.

### *IV.1.1 Contexte d'étude : projet Train-IPSat*

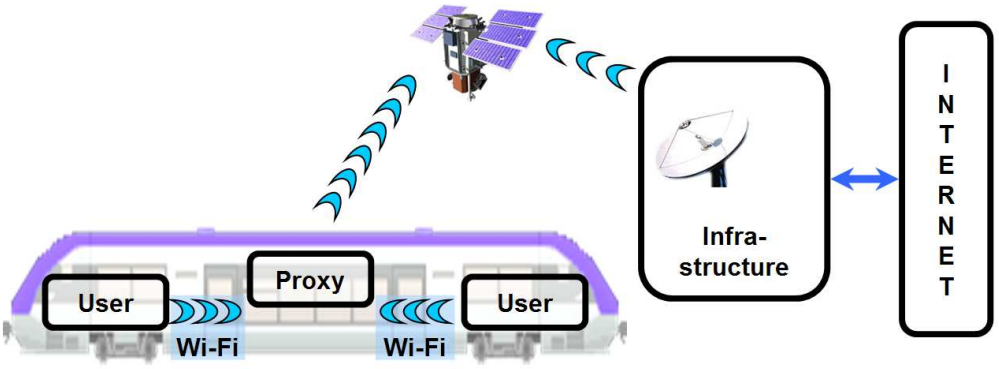


*Fig. IV.1:* Illustration du contexte de déploiement de l'application dans le projet Train-IPSat.

L'étude et le développement de ce greffon multi-modèles ont été menés dans le cadre du projet Train-IPSat [LSR+]. Ce projet vise à définir, spécifier et expérimenter des services de communication innovants offerts à la fois aux voyageurs des trains à grande vitesse pendant leur voyage et aux opérateurs ferroviaires. Dans ce contexte, notre système s'intéresse plus particulièrement à permettre un accès Internet fiable et performant aux usagers.

Pour ce projet, parmi les différentes technologies physiques de communication (GPRS, UMTS, WiMAX, Wi-Fi, Satellite), le choix a été fait d'utiliser une liaison Wi-Fi 802.11b dans le train, ainsi qu'une liaison satellitaire entre le train et l'infrastructure au sol. En effet, ces technologies permettent d'offrir le meilleur compromis entre performances, disponibilité et coût de mise en oeuvre.

Comme dans les projets actuels similaires [GL06], ici l'utilisation d'une liaison satellitaire n'est pas sans faille. Bien que le périmètre couvert par un satellite soit très important, il est impossible de garantir une liaison constante et sans interruption entre le satellite et le train, et par conséquent entre ce dernier et l'infrastructure au sol : les tunnels principalement causent des déconnexions inévitables, causant une instabilité des communications.

### *IV.1.2 Problématique et verrous*

Un système comme le prévoit le projet Train-IPSat se base sur un proxy Internet embarqué, communiquant classiquement avec l'infrastructure au sol. Le modèle de communication est donc un modèle HTTP sur TCP, standard sur Internet. Cependant ce modèle n'est pas prévu pour gérer efficacement les déconnexions induites entre autres par les tunnels : la couche IP ne permet pas de donner une solution à ces déconnexions, et la couche TCP réagit en signalant une rupture de connexion.

Le client (ainsi que le serveur) ne peut alors que signaler une erreur ; dans le cas d'un navigateur, l'usager voit alors un message du type "La connexion Internet a été perdue". E.g. dans le cas de consultations de sites Internet tels que des webmails ou gestions de comptes bancaires, si une telle déconnexion intervient, l'état de l'opération en cours (suppression, virement, envoi de message) devient indéterminé : le client est incapable de connaître l'état du serveur à ce moment.

Les propositions actuelles de bas niveau tentant de répondre aux besoins induits par la mobilité [Ern07, Gho00] présentent des systèmes de routage dynamique, mais ne fournissent aucune solution concrète aux déconnexions possibles. C'est pourquoi l'étude et le développement d'une solution de plus haut niveau peut répondre à cette problématique, et notre approche multi-modèles est bien adaptée aux verrous rencontrés.

### *IV.1.3 Développement du greffon*

En appliquant notre approche multi-modèles au problème posé, la réponse se situe évidemment plus au niveau des modèles de communication que de programmation. En effet, des systèmes de proxies existent d'ores et déjà, et présentent de bonnes performances ainsi que des fonctionnalités qui demanderaient beaucoup de travail de conception et de développement. Or le modèle de programmation est intrinsèquement lié au contexte de navigation Internet : il ne peut s'agir au final que d'une communication basée sur les Sockets TCP.

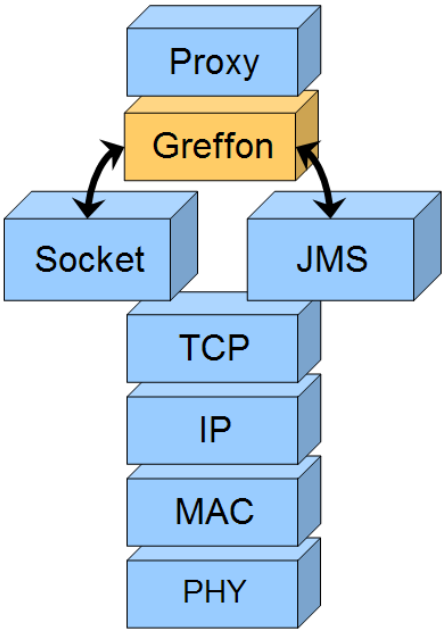


*Fig. IV.2:* Pile protocolaire résultante du proxy multi-modèles.

Comme nous le soulignons dans l'approche, le modèle de programmation peut être découplé du modèle de communication, ce qui permet une meilleure

adaptation au contexte de déploiement, sans affecter le fonctionnement de l'application. Nous proposons donc d'ajouter à un proxy Internet existant une communication multi-modèles, afin de fournir le modèle le plus adapté à chaque situation. Le modèle OSI ainsi modifié peut être schématisé comme sur la figure IV.2.

Nous allons présenter tout d'abord les modèles de communication choisis pour enrichir le fonctionnement du proxy, et les réponses qu'ils apportent au problème posé. Puis nous présenterons le système d'observation de contexte que nous avons conçu pour prendre en compte les événements extérieurs, ainsi que ses possibilités d'enrichissement. Nous détaillerons ensuite comment le comportement multi-modèles a été intégré à un proxy existant. Enfin, nous ferons apparaitre les parallèles entre la conception de ce greffon et celle de l'infrastructure multi-modèles.

*Modèles de communication utilisés*

Pour fournir au proxy le moyen de gérer les déconnexions, il est nécessaire de temporiser les communications lorsque celles-ci surviennent, pour ensuite les transmettre lorsque la liaison est rétablie. Si le modèle Socket TCP standard est incapable d'un tel comportement, un système de files d'attente de messages tel que JMS le permet tout à fait.

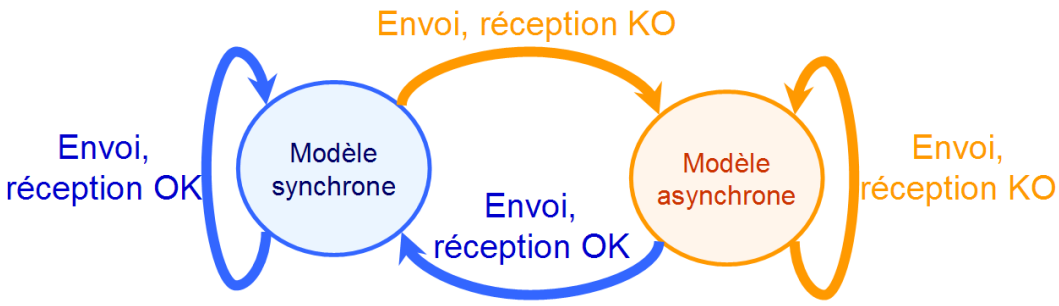


*Fig. IV.3:* Schématisation du comportement multi-modèles du greffon multi-modèles.

Le choix du modèle de communication est fait en fonction du contexte, i.e. de l'état de connexion ou de déconnexion du système. Lorsque la liaison est

établie, le modèle Socket standard est utilisé : il permet des communications rapides et directes entre le proxy et l'infrastructure au sol. Puis lorsqu'une déconnexion intervient, le modèle JMS est utilisé à son tour, afin de temporiser les messages dans une file d'attente ; ces messages seront alors transmis lorsque la liaison sera rétablie.

Cette combinaison de modèles de communication demande d'enrichir les données transmises afin de garantir une intégrité et une cohérence des transmissions. Tout d'abord, pour acheminer une communication Socket via une file de messages, les paquets de données doivent être encapsulés dans des messages [BB02]. Ces paquets, qu'ils soient transmis via l'un ou l'autre des modèles, doivent être également estampillés afin de garantir que l'ordre de réception soit le même que l'ordre d'émission, et qu'aucun paquet ne manque. Des messages particuliers sont nécessaires pour informer le récepteur de certains événements, comme une nouvelle connexion (i.e. ouverture de socket) ou une déconnexion (normale ou inopportune).

Le proxy classique doit normalement se connecter directement aux serveurs Web, cependant dans notre cas, il est nécessaire d'ajouter un serveur intermédiaire au niveau de l'infrastructure au sol afin de permettre d'une part des communications multi-modèles sur la liaison satellitaire, mais également des communications classiques avec les serveurs web. Ce serveur est chargé de reconstruire les communications avant de les faire parvenir aux serveurs, et d'envoyer les réponses via la combinaison de modèles.

*Observateurs de contexte*

**Observateurs actuellement implémentés**

Le contexte d'exécution du système de communication train-sol est assez vaste : les conditions d'utilisation des usagers, du proxy et du serveur intermédiaire peuvent varier. Cependant, l'approche multi-modèles est appliquée à la liaison entre le proxy et le serveur intermédiaire, c'est donc à ce niveau que le contexte doit être analysé. La cause la plus évidente et importante d'adaptation de cette liaison est une impossibilité totale de communiquer. L'observation de cette situation est faite régulièrement grâce à un mécanisme de ping/pong. Ce dernier tente périodiquement d'envoyer un bref message du train vers le serveur au sol, qui envoie à son tour un bref message en retour (les messages sont estampillés afin de s'assurer que la réponse correspond bien à l'émission initiale). Par ce système, le greffon connait chaque minute l'état de la liaison avec le serveur au sol.

Le mécanisme ping/pong permet d'obtenir des informations périodiques sur l'état de la liaison, mais une adaptation plus réactive aux changements de contexte ne peut qu'améliorer le système. C'est pourquoi des intercepteurs sont ajoutés dans les méthodes d'envoi et de réception Socket. Ainsi, dès qu'une exception est déclenchée dans l'une de ces fonctions à cause d'un problème réseau, le greffon en est informé, et peut alors prendre les dispositions en conséquence. Si ces intercepteurs permettent de réagir immédiatement face à une dégradation des conditions de communication, la détection de leur amélioration ne se fait que par le mécanisme ping/pong. Si celui-ci induit un certain délai (pouvant aller jusqu'à une minute), il permet surtout d'éviter des changements trop fréquents entre modèles de communication, ce qui deviendrait vite coûteux pour le système.

**Perspectives d'ajouts**

Pour obtenir des informations sur la qualité du lien réseau, qui soient plus directes et surtout plus précises, nous avons prévu d'integrer un observateur d'événements utilisant l'interface libpcap[1]. Cet observateur, en plus de fournir un état élémentaire du réseau (en service, ou hors service), permet de diagnostiquer plus précisément la qualité de la liaison, e.g. en contrôlant l'intégrité des paquets via les contrôles d'erreurs IP et TCP. Une perte progressive d'intégrité pouvant présager une rupture prochaine de la liaison, le greffon pourrait alors adapter son comportement avant même que l'événement ne se produise.

Enfin, dans le contexte des transports ferroviaires, les itinéraires ainsi que les horaires réguliers permettent d'établir des prévisions de disponibilité des satellites en fonction de la position du train et de l'heure [MLB04]. En associant un récepteur GPS à une telle prédiction, le greffon pourrait réagir encore plus précisément au contexte, car en complément de l'observation directe des paquets de données circulant sur le réseau, il serait en mesure de connaitre exactement l'instant où les déconnexions et reconnexions se produiront.

Bien que ces perspectives plus évoluées d'observateurs de contexte soient envisageables, le système actuel permet néanmoins d'utiliser le meilleur modèle de communication en fonction de l'état de la liaison, même si le temps de rétablissement d'un lien synchrone peut prendre jusqu'à une minute. Les perspectives permettraient de réduire (voire de supprimer) ce délai, rendant le système encore plus performant qu'il ne l'est déjà.

[1] Packet Capture Library, bibliothèque de capture de paquets bas niveau sur le réseau, `http ://www.tcpdump.org`.

*Ajout du comportement multi-modèles à l'application existante : ASM*

L'objectif de cette application de notre approche est de permettre à n'importe quelle application déjà existante d'adopter un comportement multi-modèles. L'une des problématiques est donc de permettre à notre greffon de s'adapter à des applications déjà développées (et donc compilées), sans que le code source de celles-ci ne soit forcément disponible et modifiable. Nous avons ainsi choisi de modifier directement les méthodes appelées par ces applications pour gérer leurs communications : la classe `Socket` de la JVM[1].

Il a été nécessaire dans un premier temps d'identifier les méthodes concernées par l'intervention du greffon. Celles-ci lui permettent de se substituer naturellement et efficacement au fonctionnement du Socket classique de Java. Une fois identifiées, celles-ci ont été surchargées pour intégrer les mécanismes d'adaptation multi-modèles : les méthodes de création (constructeur), de connexion (`connect`) et de transfert (`getInputStream` et `getOutputStream`) permettent alors aux communications d'être interceptées.

Pour surcharger ces méthodes automatiquement, et ce indifféremment de la version de la JVM utilisée, nous avons développé une application chargée de fournir un paquetage composé d'un ensemble de classes comprenant la version modifiée du `Socket`. Pour intervenir directement dans le bytecode existant de la JVM, nous avons utilisé ASM [BLC02], un outil de modification de bytecode mis à disposition par le consortium ObjectWeb. Une fois le paquetage généré, son utilisation en lieu et place du `Socket` standard nécessite simplement de spécifier son nom en ligne de commande (i.e. `java -Xbootclasspath/p:greffon_amm.jar nomDuProxy`). En suivant ce principe, si l'application ne nécessite plus l'utilisation du greffon, il suffit de supprimer le paramètre en ligne de commande.

*Comparaison avec l'infrastructure logicielle multi-modèles : une architecture globale commune*

Même si le greffon n'est qu'une application basique de l'approche multi-modèles, vouée à justifier et démontrer son intérêt, il n'en est pas moins une version allégée de l'infrastructure logicielle multi-modèles. En effet, on retrouve ici la même structure centralisée, présentant plusieurs composants complémentaires :

[1] Java Virtual Machine, machine virtuelle Java permettant d'intepréter et donc d'exécuter les classes compilées des applications.

- Le modèle de programmation présenté est une classe `Socket` standard ; le programme n'a ainsi aucun besoin de modifier lui-même son mode de communication.
- Des observateurs de contexte fournissent des informations relatives à la qualité du réseau ; ils sont similaires aux observateurs décrits dans l'infrastructure.
- Bien qu'il n'y ait aucune politique d'adaptation paramétrable, les mécanismes suivent des règles simples : si les observations sont concluantes (i.e. succès du ping/pong et aucune exception anormale du modèle `Socket`), le modèle `Socket` est utilisé, sinon le greffon utilise JMS.
- Les modèles de communications sont combinés et hiérarchisés, permettant l'utilisation du `Socket` en priorité lorsqu'il est actif, puis de JMS le cas échéant.
- Un composant central (un "*Manager*") est chargé de convertir les requêtes du modèle de programmation (les paquets de données soumis au `Socket`) en des communications via les différents modèles de communication (i.e. des paquets `socket` enrichis, et des messages JMS).

### *IV.1.4 Analyse et comparatif des performances*

Afin de justifier l'intérêt de notre approche, nous avons effectué grâce au travail de Franck Desbuquois et Julien Merlin [DM08] des tests comparatifs mettant en situation le proxy embarqué. Dans un premier temps, le greffon n'a pas été utilisé, pour obtenir des résultats témoins, représentatifs des performances affichées par un système standard de proxy. Puis le greffon a été implanté sur le proxy ; son fonctionnement a été dirigé pour présenter des résultats exploitables. En effet, le contexte des tests est volontairement stable, pour ne pas biaiser les comparaisons. Le greffon a donc été utilisé avec son modèle de communication synchrone, puis asynchrone.

#### *Configuration des tests*

La configuration choisie pour effectuer les tests est représentative du contexte de déploiement du projet Train-IPSat ; les scénarii d'utilisation ont également été pensés pour simuler une utilisation "classique" de l'accès à Internet.

**Configuration matérielle**

La plateforme d'exécution des tests est un PC portable Dell XPS M1210, équipé d'un processeur Intel T2500 Core Duo à 2GHz et de 1Go de mémoire vive. La seconde plateforme d'exécution dédiée au serveur web et au serveur multi-modèles est un PC portable Dell Latitude C840 équipé d'un processeur

Intel Pentium 4-M à 2,4GHz et de 784Mo de mémoire vive.
La liaison entre les deux plateformes est assurée par un lien réseau sans fil 802.11b présentant un débit moyen de 800Ko/s, débit comparable aux possibilités offertes par une liaison satellitaire. Le lien entre le programme de tests et le proxy est assuré par la boucle interne de la plateforme : les tests sont chargés d'évaluer les performances du lien proxy-infrastructure, peu importe si les communications en amont du proxy ne sont pas réalistes, elles doivent juste être constantes en termes de performances.

**Configuration logicielle**
Nous avons choisi JMeter[1] pour exécuter les scénarii et recueillir les mesures de performances. Le serveur proxy utilisé lors des tests et le même que pour le développement du greffon, i.e. Muffin[2], un proxy web développé en Java. Le serveur Web sur lequel les requêtes sont posées est le serveur Apache. Les plateformes utilisent des systèmes d'exploitation Microsoft Windows et la machine virtuelle Java est une JVM 1.5.0 de Sun. Bien que ces tests se soient déroulés sous un environnement Windows, des tests similaires ont été menés sous un environnement Debian (i.e. Linux) et présentaient des résultats équivalents.

**Scénarii de tests**
Les scénarii conçus pour ces tests ont été conçus pour représenter l'utilisation typique des utilisateurs :

- Une recherche par mots-clés sur un moteur de recherche (Google™).
- Une consultation de portail généraliste (Yahoo !®).
- Un visionnage de vidéo en ligne (`http ://www.koreus.com`).
- Une consultation de courriels en ligne dans une interface dynamique (ZK® Mail).

Pour chacun des scénarii, les requêtes sont effectuées 100 fois chacune afin de lisser les résultats et d'obtenir une moyenne plus globale des résultats obtenus.

Pour des raisons de constance de contexte, toutes les requêtes sont effectuées sur un serveur web local afin d'éviter que la charge fluctuante du serveur ou la bande passante variable de l'accès Internet n'influent sur les résultats.

[1] JMeter, fait partie du projet Apache Jakarta, `http ://jakarta.apache.org/jmeter`
[2] Muffin, "World Wide Web Filtering System", `http ://muffin.doit.org`

*Résultats des test*

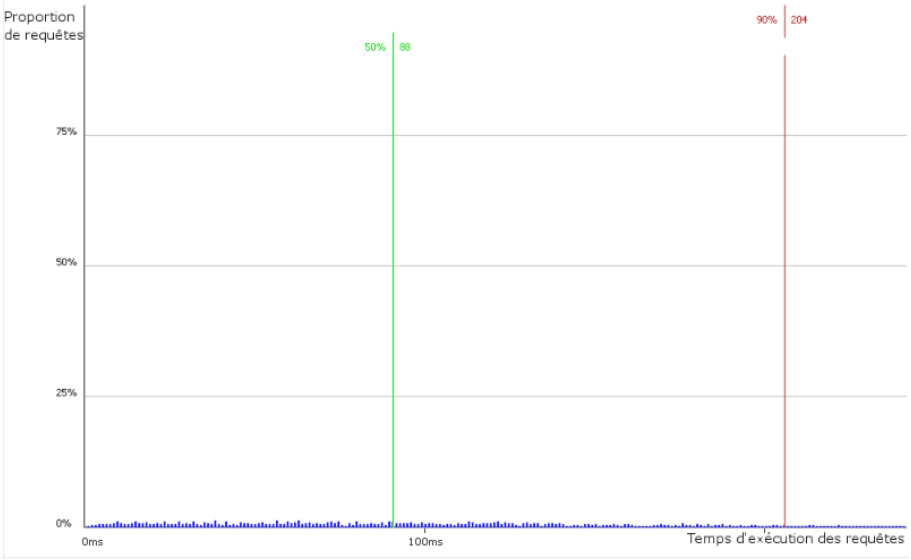


*Fig. IV.4:* Graphique de distribution des temps de requêtes via le proxy sans greffon.

Lors de la première exécution de tests, le proxy est utilisé seul. Les résultats représentent donc les performances atteignables avec un système de proxy traditionnel. Le graphique de répartition en figure IV.4 indique que la moitié des requêtes ont été effectuées en moins de 88ms, et 90% des requêtes se sont terminées en moins de 204ms.

Le serveur Web étant local, et les contenus étant tous des fichiers statiques (y compris pour les données originalement dynamiques comme les Php, Jsp, Asp, cgi), ceci explique la moyenne plutôt basse des résultats obtenus.

La seconde exécution des tests concerne le proxy équipé du greffon. Ce dernier est utilisé en conditions réelles (i.e. il est susceptible de changer de modèle de communication si une déconnexion intervient), mais la liaison étant continue entre le greffon et le serveur multi-modèles, aucun changement de modèle n'intervient. Le modèle de communication utilisé est donc le tunnel `Socket`.

La figure IV.5 représente la répartition des temps de transmission et d'exécution des requêtes. La moitié de celles-ci ont été effectuées en moins de 94ms,

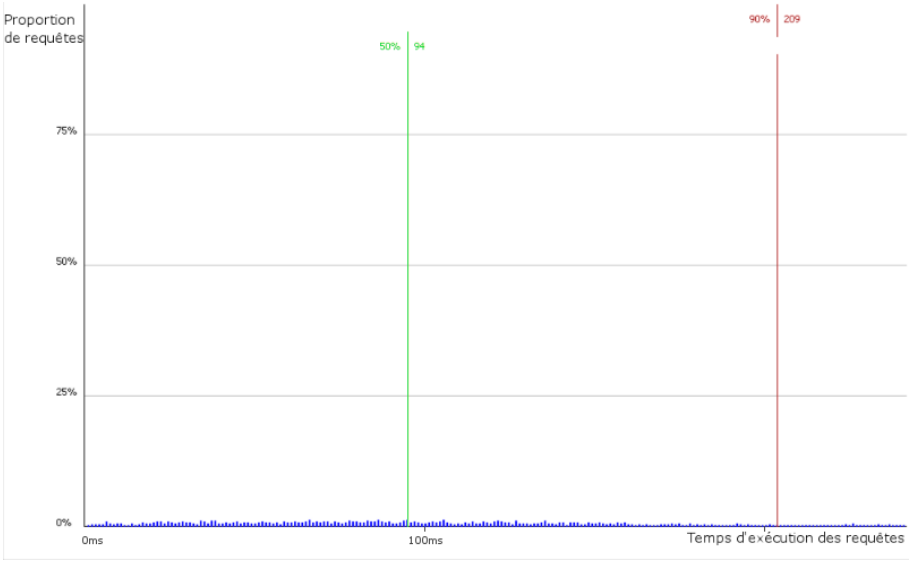


*Fig. IV.5:* Graphique de distribution des temps de requêtes avec le greffon en mode Socket.

et 90% d'entres elles l'ont été en moins de 209ms. La différence entre l'utilisation du greffon en `Socket` et le proxy seul est faible, mais néanmoins existe. Après plusieurs tests successifs exécutés pour vérifier ces données, la différence est toujours de cet ordre, avec des variations allant jusqu'à 15ms pour le temps médian, et jusqu'à 25ms pour les 90% des requêtes. Le graphe représenté ici représente la moyenne des tests effectués.

La troisième exécution des tests a permis de mesurer les performances du proxy utilisant le greffon forcé en mode JMS. En effet, pour juger des différences entre le système multi-modèles et le système original, il nous a semblé judicieux de mesurer le meilleur cas (utilisation continue du tunnel `Socket`), ainsi que le pire ("Worst Case Scenario" : 100% JMS).

La figure IV.6 montre clairement la différence entre l'utilisation de JMS et les autres cas d'utilisation. La médiane du temps des requêtes se situe à 135ms, et l'autre valeur remarquable est trop élevée pour figurer sur le graphique : 90% des requêtes ont été traitées en moins de 307ms.

### *Comparatifs et bilan*

D'un point de vue objectif, chiffres à l'appui, le greffon multi-modèles offre des temps d'exécution légèrement supérieurs au système standard : l'utilisa-

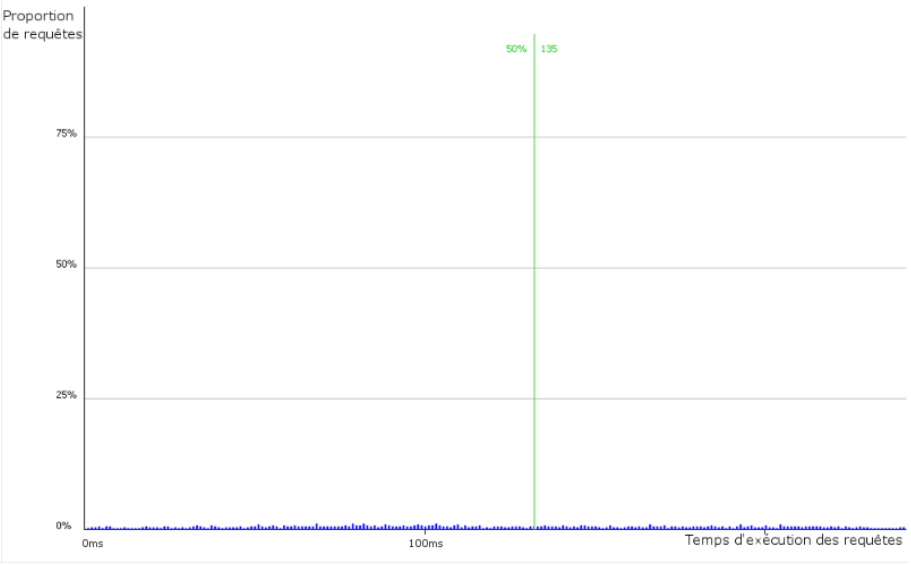


*Fig. IV.6:* Graphique de distribution des temps de requêtes avec le greffon en mode JMS.

tion exclusive du modèle `Socket` provoque un ralentissement global du temps d'acheminement et de traitement des requêtes de 2,5 à 7% (selon l'indicateur statistique pris en compte) ; alors que l'utilisation exclusive du modèle JMS pousse ce ralentissement à un pourcentage situé entre 50 et 54%.

Il est maintenant nécessaire de replacer ces chiffres dans leur contexte. Si l'on ne doit retenir qu'une donnée pour résumer ce comparatif de performances, ce doit être le ralentissement provoqué par le greffon en mode `Socket`, soit une valeur avoisinant les 5%. La raison en est simple : l'utilisation standard d'un proxy n'est possible qu'en contexte stable, i.e. la connexion est établie en continu. Le cas échéant, des exceptions et erreurs interviendront et réduiront la qualité des communications (dans le meilleur des cas). Le greffon multi-modèles dans ce même contexte stable n'utilise alors que son modèle `Socket`, leurs performances peuvent donc bien être comparées. Par contre l'utilisation du modèle JMS par le greffon signifie que le contexte devient instable. Le système classique dans ces conditions ne pourrait assurer aucune communication ; les performances du greffon en mode JMS doivent donc être comparées au système dans un contexte équivalent, c'est-à-dire un système inutilisable. Dans ces conditions, même si le modèle JMS induit des ralentissements par rapport au modèle `Socket`, il n'en reste pas moins la seule solution pour assurer des communications en contexte instable.

## *IV.2 L'infrastructure logicielle multi-modèles : implémentation concrète des éléments de l'approche*

Par l'étude et le développement du greffon multi-modèles adapté au proxy Internet embarqué, nous avons montré et démontré des principes fondamentaux d'une infrastructure multi-modèles, ainsi que leurs intérêts en terme de gain de sécurité de communication. Cependant, le modèle de programmation utilisé par notre greffon, i.e. l'utilisation d'un `Socket` basique, ne propose pas un éventail très étendu de possibilités de programmation, et les différents aspects importants de notre infrastructure ne sont pas exploités et implantés pleinement : e.g. les politiques d'adaptation sont réduites à un choix élémentaire. Nous avons donc entrepris la conception et la réalisation d'une infrastructure logicielle multi-modèles complète, c'est-à-dire présentant tous les différents éléments principaux de notre approche.

Chacun de ces éléments présente son intérêt spécifique dans notre approche multi-modèles. Il est donc nécessaire d'une part de concevoir simplement chaque élément, i.e. d'assurer ses fonctions élémentaires, sans forcément chercher à assurer exhaustivement le fonctionnement optimal dans toutes les situations. Il est d'autre part également nécessaire de permettre des extensions à ces éléments (e.g. patrons de conception additionnels, ajouts de possibilités aux politiques d'adaptation, changements ou amélioration du comportement du manager).

Les deux implémentations que nous présentons ici ne fournissent pas toutes les possibilités que notre approche permet d'envisager. Dans un premier temps nous les avons conçues et développées afin de prouver que notre approche et viable et profitable. Basées sur les points clés de notre approche, nous détaillerons dans un premier temps une version statique de notre infrastructure, intégrant le comportement multi-modèles dans des mécanismes établis lors de la compilation de l'application, puis nous détaillerons dans un deuxième temps une version dynamique, utilisant la modification de bytecode à l'exécution et de l'adaptation dynamique pour rendre l'application plus réactive au contexte.

Les infrastructures logicielles multi-modèles résultant de ces implémentations démontrent le bien fondé et l'utilité de notre approche. Leur amélioration et leur enrichissement au travers de fonctionnalités supplémentaires et de perfectionnements pourra permettre d'envisager leur application à un

panel plus élargi de situations et d'utilisations, justifiant d'autant plus la légitimité de notre démarche.

### *IV.2.1 Étude de faisabilité : les mécanismes d'adaptation*

La combinaison et la communication entre les différents éléments de l'infrastructure multi-modèles engendrent certains mécanismes internes qui nécessitent une attention particulière. En effet pour pouvoir présenter un fonctionnement clair et performant indépendamment du modèle de programmation et de communication choisi, tout en autorisant une adaptation efficace aux changements de contexte, nous avons dû définir les mécanismes suivants :

- L'objet `MultiModelInvocation` fournit un format de transmission d'invocation complet et transportable (`Serializable`). Comme expliqué au chapitre III.7, cet objet permet à l'émission de spécifier le type d'invocation, les références de l'appelant et de l'appelé, et les paramètres à transmettre à l'appelé. Pour le trajet retour de l'invocation, l'objet contient de plus la valeur de retour ou l'exception déclenchée par l'invocation.
- Un système d'estampillage des invocations est également ajouté, afin d'identifier sans équivoque chaque appel ou communication. Ce mécanisme évite ainsi les invocations multiples issues d'un même appel (lors d'utilisation de plusieurs modèles de communication par exemple), et assure la récupération de la valeur de retour correspondant à une invocation, même en cas de changement de modèle.

#### *Emulation des paradigmes et patrons de conception*

Comme nous l'avons décrit dans la section III.2.4, le modèle de programmation de notre infrastructure propose d'une part un modèle RPC, et d'autre part des patrons de conception permettant d'utiliser des files de messages, sujets d'intérêts et autres espaces partagés. Si le modèle RPC, fortement couplé, dépend du fil d'exécution du client (i.e. l'émetteur de l'invocation), la programmation répartie asynchrone introduit un nouveau fil d'exécution, indépendant du client et du serveur. En effet, lorsqu'un composant émet une requête asynchrone, son exécution est dissociée de celle de la requête.

C'est pour cette raison que l'objet `MultiModelInvocation` présente une deuxième fonctionnalité : la prise en charge de l'invocation pour le client. Si cet objet présente des données utiles transmises tout au long de l'acheminement de l'invocation (grâce à l'implémentation de l'interface `serializable`),

il présente également des méthodes utilisées par les objets proxy, spécifiques à chaque type d'invocation.

```
public interface MultiModelInvocationItf extends java.io.Serializable
{
        public InvocationType getType() ;

        public MultiModelReference getSourceObject() ;
        public MultiModelReference getTargetObject() ;
        public String getTargetMethod() ;
        public Object[] getParameters() ;
        public long getInvocationID() ;

        public Object getReturnValue() ;
        public Throwable getErrorOrException() ;

        public long[] getPendingRequests() ;
        public void mergeReturnInvocation(
                MultiModelInvocationItf invocationFromServer) ;
}
```

*Fig. IV.7:* Interface respectée par la classe `MultiModelInvocation` de l'infrastructure.

Pour chacun des types d'invocation proposés, une classe héritée de `MultiModelInvocation` fournit en complément des méthodes de gestion de l'invocation :

- `SynchronousInvocation` permet l'appel d'une méthode `Object invoke() throws Throwable`.
- `AsynchronousInvocation` présente la méthode `void invoke()`, ainsi que deux méthodes `boolean isReturnValueAvailable()` et `Object getReturnValue() throws Throwable`.
- `OneWayInvocation` quant à elle propose la seule méthode `void invoke()`.

Les patrons de conception fournis par l'infrastructure utilisent l'une ou l'autre des classes proposées afin d'acheminer l'invocation la plus adaptée à leur fonctionnement, e.g. une file de message utilise une `OneWayInvocation`.

L'utilisation d'invocations asynchrones induit un fil d'exécution indépendant du client et du serveur, comme nous l'avons souligné plus tôt. Le Manager doit donc utiliser un `Thread` indépendant pour gérer ce type d'invocation. Si le problème n'a pas été soulevé dans l'implémentation actuelle de

l'infrastructure, une gestion efficace de ces `Thread` doit néanmoins prévoir des montées en charge importantes : l'utilisation d'une batterie de `Threads` de taille prédéfinie permettrait d'optimiser ce type d'invocations, en évitant des créations/suppressions coûteuses et inutiles de ces `Threads`.

### *IV.2.2 Implémentation du modèle de programmation*

Comme nous l'avons expliqué au chapitre III.2, un seul type de programmation ne peut suffire au développeur pour lui permettre un développement efficace d'applications distribuées. Nous proposons donc trois types de programmation répartie ; un premier basé sur les annotations, un second similaire aux mécanismes d'invocation dynamique de certains intergiciels, et un troisième s'appuyant sur des patrons de conception.

Si la finalité de ces types de programmation est commune (i.e. la création et la gestion d'une `MultiModelInvocation`), leur forme particulière optimise la programmation des différents modèles de répartition. Ainsi par exemple les annotations simplifient grandement la programmation en général, mais ne permet pas d'utiliser complètement tous les modèles de répartition (cf section suivante).

*Annotations basiques*

```
public class Client {
              @ResolveObject(
       name = "*server*" ;
       location = null ; )
              private Server myServer ;

              public Client() {
              /* Nothing to do here */
              /* Automatic resolve */
              }

              public void doSomething() {
              /* implicit remote call */
              myServer.doSomething() ;
              }
}
```

```
@DistributedClass( type = { "Sync" } ; )
public class Server {
              @DistributedObject(
                            name = "server_001" ;
                            registered = true ; )
              public static Server = new Server() ;

              public Server() {
              /* Initialisation stuffs */
              }

@DistributedMethod
              public void doSomething() {
              /* do … something */
              }
}
```

*Gris et italique = paramètres optionnels, valeurs par défaut*

*Fig. IV.8:* Exemple de code d'application annotée.

Nous avons défini un certain nombre d'annotations qui permettent une programmation simple d'applications réparties, décrites en annexe A. Elles permettent principalement d'offrir un modèle de programmation de type RPC simple et efficace. Si pour les composants clients ces annotations permettent également d'offrir d'autres modèles de programmation, c'est en s'associant à l'utilisation des patrons de conception que nos proposons (cf section suivante).

En ce qui concerne les composants serveurs, les annotations permettent certains modèles de programmation, mais avec certaines restrictions : un objet peut se déclarer lui même comme file de messages ou sujet d'intérêt (précisé par l'annotation `DistributedClass`), mais il sera alors obligatoirement considéré comme le consommateur unique.

*Patrons de conception*

En complément du modèle de programmation par annotations, l'infrastructure logicielle multi-modèles fournit plusieurs patrons de conception, sous la forme de classes. Pour utiliser l'un de ces patrons, l'instanciation se fait indifféremment via une annotation ou un appel explicite, comme l'illustre la figure IV.9.

```
/* Exemple d'utilisation d'une file de messages
avec les annotations */
public class AnnotatedPublisher {
            @ResolveObject(
      name = "MaFile" ;
      location = "localhost" ; )
            private MessageQueue myQueue ;

            public Publisher() {
            /* Nothing to do here */
            /* Automatic resolve */
            }

            public void doSomething() {
            /* implicit remote call */
            myQueue.send(new Message(null)) ;
            }
}
```

```
/* Exemple d'utilisation d'une file de messages avec
les méthodes Factory */

public class FactPublisher {
            private MessageQueue myQueue ;

            public Server() {
            myQueue = Manager.MessageQueueFactory()
            .resolveObject("MaFile","localhost") ;
            }

            public void doSomething() {
            myQueue.send(new Message(null)) ;
            }
}
```

*Gris et italique = paramètres optionnels, valeurs par défaut*

*Fig. IV.9:* Instanciation d'une file de messages par annotation et factory.

Les différents patrons de conception permettent ainsi l'utilisation de communications basées sur les files de messages (`MessageQueue`), les sujets d'intérêt (`Topic`), les événements (`Events`, non encore implémentés) et les espaces

partagés (`SharedSpace`). Chaque objet ainsi fourni propose des méthodes dédiées à son type de communication, e.g. `MessageQueue` permet des appels aux méthodes `Message receive()`, `boolean isMessageAvailable()`, `void send(Message msg)`, ainsi que des méthodes communes d'identification et de gestion des droits (`void connect(String id, String password)`, `void disconnect()`, `void addUser(String id, String password, boolean canRead, boolean canWrite, boolean canAdmin)`, `void removeUser(String id)`).

*Invocation explicite : DII et DSI*

Comme certains intergiciels actuels le proposent, notre infrastructure permet au développeur de dynamiquement construire les invocations, et rendre accessibles des méthodes d'un objet réparti. Dans le premier cas, les mécanismes d'invocation dynamique sont accessibles via une interface (DII) qui est identique à celle utilisée par les objets proxies. Le développeur utilise une usine pour créer un objet `MultiModelInvocation` sur lequel il peut ensuite appeler les méthodes spécifiques au type d'invocation,
e.g. `Manager.InvocationFactory().createOneWayInvocation( MultiModelReference remoteObject, String method, Object[] parameters).invoke();`.

Dans le deuxième cas, un objet annoté peut être rendu accessible via un simple appel de méthode `Manager.provideObject(Object serverObject)`, et dans tous les cas, qu'il ait été rendu accessible par une annotation ou un appel explicite, il peut se rendre inaccessible grâce un appel à `Manager.unprovideObject(Object serverObject)`.

### *IV.2.3 Prise en charge des annotations au chargement*

L'application développée grâce à l'infrastructure logicielle multi-modèles se présente, une fois compilée, sous la forme d'une application standard, à l'exception près que chaque classe communicante décrit son fonctionnement réparti via des annotations. L'infrastructure a donc besoin d'intervenir au chargement de l'application, pour localiser et prendre en compte ces annotations.

Nous avons choisi d'utiliser le principe d'"agent d'instrumentation" de Java [Mic04] afin de superviser le chargement des classes de l'application. En fournissant à la JVM un paquetage Jar contenant l'infrastructure logicielle multi-modèles et spécifiant quels agents doivent être consultés au chargement

du programme, ce principe permet d'ajouter des transformateurs de classes (`java.lang.instrument.ClassFileTransformer`) capables de modifier le bytecode de toutes les classes chargées par la JVM.

Pour modifier la classe, le transformateur ne dispose que de son bytecode. Comme nous l'avons déjà utilisé lors du développement du greffon multi-modèles, nous avons choisi ASM pour décoder et modifier ce bytecode.

Le transformateur que l'infrastructure désigne analyse donc dans un premier temps les classes chargées, et détecte la présence des annotations qu'elle fournit grâce à ASM, notamment `ResolveObject` et `DistributedClass` qui provoquent la prise en charge de la classe par l'analyseur.

Cet analyseur s'intéresse tout d'abord à la concordance des annotations présentes dans la classe, en l'absence de quoi la classe sera ignorée. Différents modes d'exécution de l'infrastructure existent, et attachent plus ou moins d'importance aux erreurs de cette prise en charge des annotations : e.g. certains déclenchent une `MultiModelAnnotationCoherenceException` – le chargement de l'application est donc interrompu –, alors que d'autres affichent juste un message d'erreur (ou n'affichent rien) et continuent le chargement.

La concordance est définie selon des conditions précises. Ce dernier vérifie que l'utilisation d'une annotation conditionnée est faite en présence des conditions définies (e.g. l'annotation conditionnée `QueueReceiveMethod` doit être déclarée une et une seule fois, et ce exclusivement dans une classe annotée `DistributedClass(type="Queue")`).

Lorsque la concordance est positive entre les annotations d'une classe fournie au `ClassFileTransformer`, le bytecode de cette classe peut alors être modifié pour utiliser l'infrastructure logicielle multi-modèles. Deux cas de figure se présentent alors, l'un concernant les classes annotées `DistributedClass`, et l'autre concernant les classes possédant un attribut annoté `ResolveObject`.

- Dans le premier cas, le transformateur ne modifie pas directement le code de la classe, mais définit une nouvelle classe "proxy serveur" héritant de celle-ci, et assurant le rôle d'intermédiaire entre le Manager et les modèles de communication d'une part, et la classe concernée d'autre part. Seules les méthodes annotées `DistributedMethod`

sont ainsi surchargées ; une méthode `__MultiModelInvoke` et des méthodes `get` et `set` correspondantes aux attributs annotés `DistributedAttribute` sont également ajoutées. Cette première méthode est invoquée lorsque le client et le serveur présentent un comportement multi-modèles, elle permet de fournir au serveur l'objet MultiModelInvocation du client.

- Dans le deuxième cas, le transformateur modifie la classe afin d'assurer la communication avec l'objet réparti ; la classe "proxy client" du ou des objets répartis utilisés est générée – si elle n'a pas encore été définie – sur le même principe que celle du "proxy serveur" (et utilise une `MultiModelInvocation` pour acheminer l'invocation). Le constructeur de la classe concernée est également modifié pour instancier un ou des "proxies client". De plus, le corps des méthodes est modifié dans deux cas :
  - Lorsque les attributs de l'objet réparti sont utilisés, l'accès direct est remplacé par l'utilisation de méthodes `get` et `set` afin d'être cohérent avec le "proxy serveur" correspondant, et de permettre l'utilisation répartie des attributs via des invocations.
  - Lorsque les appels de méthodes sont de type asynchrone, on utilise alors le système d'optimisation du code décrit par la suite.

**Le système d'optimisation du code d'appel et ses limites**

Le modèle de programmation générique propose des mécanismes d'invocation asynchrone, permettant dans un premier temps d'utiliser un objet `AsynchronousInvocation` pour construire dynamiquement et explicitement une invocation par le biais de l'infrastructure multi-modèles. Le développeur peut ainsi déclencher cette invocation sans pour autant bloquer le fil d'exécution de l'appelant, et utiliser la valeur de retour de l'invocation lorsque celle-ci est disponible soit en scrutant son arrivée, soit en demandant cette valeur par le biais d'un appel à une méthode spécifique de l'`AsynchronousInvocation`. Ce concept de *Future* [Hal85], s'il permet une grande souplesse dans la programmation asynchrone, ne permet cependant pas particulièrement d'optimiser l'écriture d'une application répartie.

Les annotations permettent également au développeur de profiter de cette programmation asynchrone. Ainsi lorsque la classe de l'objet distant est définie comme entièrement asynchrone (`@DistributedClass(type="Async")`), qu'une instance de cette classe le soit (`@DistributedObject(type="Async")`), ou bien encore que l'une de ses méthode le soit (`@DistributedMethod(type="Async")`), dans tous ces cas le code appelant la méthode est modifié pour

appeler une méthode dédiée de l'objet "proxy client" nommée `AsynchronousInvocation __asyncInvoke(String method, Object[] parameters)`. Cette méthode permet ensuite de ne demander la valeur de retour que lorsque le code de l'appelant utilise réellement celle-ci (i.e. lorsque la variable stockant la valeur de retour est utilisée). Ce type de modification n'est envisageable qu'en cas d'utilisation d'une variable locale pour le stockage de cette valeur, car l'utilisation d'un attribut ou de tout autre variable susceptible d'être utilisée par plusieurs fils d'exécution serait complexe voire impossible à mettre en œuvre.

Le système d'optimisation du code d'appel ne s'adresse donc qu'aux classes annotées dans deux cas précis :

- Lors de l'utilisation d'une méthode définie comme "asynchrone" en stockant la valeur de retour dans une variable locale et en ne l'utilisant pas directement au moment de l'appel (e.g. une utilisation comme `proxyClient.methodeDistante().toString()` interdit toute utilisation de *Future*). L'invocation est bien envoyée immédiatement, mais la requête de la valeur est retardée.
- Lors de l'utilisation d'une méthode définie comme "unidirectionnelle". L'invocation est bien également envoyée immédiatement, mais le code n'attend pas que l'objet distant ait complété la requête pour continuer son exécution.

Dans l'état actuel des choses, aucune optimisation n'est effectuée lors de l'utilisation d'un objet "proxy client" créé autrement que par une annotation `ResolveObject` ou lorsqu'une méthode externe à l'objet possédant l'objet proxy appelle une méthode sur ce dernier. L'appelant doit dans ces cas utiliser le Manager pour permettre une utilisation asynchrone des méthodes (en appelant `AsynchronousInvocation Manager.doAsynchronousInvocation(Object remoteObject, String method, Object[] parameters)`), et utiliser lui-même explicitement l'`AsynchronousInvocation` ainsi générée.

L'optimisation de ce code pourrait également être effectuée, ce qui rendrait la programmation dynamique aussi performante que la programmation par annotations, cependant l'identification de tous ces cas est complexe : un même attribut ou objet local peut tour à tour être un objet local ou un proxy client, et modifier toutes les classes pour insérer une vérification du type de l'objet (i.e. utilisation de la méthode `proxyClientClass.isAssignableFrom(object.getClass())`) serait très coûteux et pénaliserait l'entière exécution de l'application.

### *IV.2.4 Observateurs de contexte*

```
public class RandomContextObserver extends java.util.Observable
{
            /** Méthode fournissant une observation de contexte */
            public double getValue() {
                        /* Le Manager aura appelé clearChanged() */
                        /* avant getValue(). Il faut donc changer l'état */
                        /* de l'observable pour que le Manager consulte */
                        /* de nouveau la méthode au prochain cycle */
                        this.setChanged() ;
                        /* Retourne une valeur aléatoire */
                        return Math.random() ;
            }
}
```

*Fig. IV.10:* Exemple de classe d'observateur de contexte.

Les observateurs de contexte héritent tous d'une même classe `java.util.Observable`. Chaque observateur est un singleton instancié par le Manager lorsque les politiques d'adaptation en font mention (i.e. lorsqu'un observateur est rencontré dans les politiques, s'il n'a encore jamais été utilisé, le Manager l'instancie).

L'héritage de la classe `Observable` permet aux observateurs d'offrir deux modes d'utilisation : soit ils préviennent lorsque leur état a changé (appel de la méthode `notifyObservers`), soit dans tous les cas le Manager effectue un contrôle régulier de l'état de ses observateurs (consultation par la fonction `hasChanged()`). Chaque observateur peut donc être actif ou passif.

Chaque observateur doit présenter un nom de méthode ou un nom d'attribut correspondant à la description utilisée dans les politiques, e.g. si une politique dépend de la condition

```
<Cause>
    <Observation type="Attribute" data-type="int">
        WifiSignalObserver.signalLevel
```

```
    </Observation>
    <Criterion type="<=">25</Criterion>
</Cause>
```

alors l'observateur doit posséder un attribut `int signalLevel`.

Nous avons implémenté selon ce principe l'observateur développé pour le greffon multi-modèles. Il en résulte un observateur nommé `PingPongObserver` fournissant pour les politiques un simple attribut `isConnected`. Cet observateur est par ailleurs configurable par le développeur afin de lui fournir les paramètres adéquats (via les méthodes statiques `setHostAddress`, `setDelay`, `setTimeout`).

*Observateurs liés aux modèles de communication*

Nous avons également développé un observateur correspondant à chaque modèle de communication utilisé, appelés `RMIObserver` et `JMSObserver`. Chacun présente des méthodes et attributs spécifiques à son modèle afin de fournir les observations les plus pertinentes possibles.

Chaque observateur propose une méthode `boolean canAccessToNaming( String host, int port)` qui vérifie si le service de nommage peut être joint ou non. Si l'infrastructure utilise plusieurs services différents (spécifiés par le paramètre `location` de l'annotation `ResolveObject`), le Manager testera chaque registre utilisé et la politique concernée ne s'appliquera qu'aux communications originaires de ce registre (en utilisant pour le `Result` le type `"DisableUseOfModel"`), ou bien dès qu'un registre est injoignable le modèle n'est plus du tout utilisé (en utilisant le type `"DisableModel"`, cf section IV.2.5).

En complément, chaque observateur relatif à un modèle de communication fournit une liste de références d'objets pour lesquels une exception a eu lieu lors d'une invocation. Cette liste nommée `List invocationExceptions` est mise à jour en temps réel dès qu'une exception intervient dans le modèle concerné, et bien que le fonctionnement passif de l'observateur soit ici possible, notre choix a été d'invoquer la méthode `notifyObservers` immédiatement, pour assurer une bonne réactivité du système. Similairement à la méthode `canAccessToNaming`, la politique utilisant cet attribut peut ne concerner que les communications pour un objet distant, ou bien l'intégralité de l'utilisation de ce modèle de communication.

### *IV.2.5 Prise en charge des politiques en temps réel*

```
<Policy>
 <Cause>
  <Observation type="Method" data-type="double">
   RandomContextObserver.getValue
  </Observation>
  <Criterion type="<">0.5</Criterion>
 </Cause>
<Result type="Disable"><Model>RMI</Model></Result>
</Policy>
<Policy>
 <Cause>
  <Observation type="Method" data-type="double">
   RandomContextObserver.getValue
  </Observation>
  <Criterion type=">=">0.5</Criterion>
 </Cause>
<Result type="Enable"><Model>RMI</Model></Result>
</Policy>
```

*Fig. IV.11:* Politique d'adaptation prenant en compte l'observateur décrit en figure IV.10.

Pour permettre au Manager de prendre en compte les politiques d'adaptation dès qu'elles sont actualisées, un `Thread` indépendant se charge de "surveiller" l'état du fichier de politiques, afin de réagir dès qu'il est modifié (vérification de sa date de dernière modification pour connaître son état).

Les politiques d'adaptation peuvent ensuite être interprétées afin de construire une chaine de `AdaptationPolicy` correspondant au contenu du fichier. Chaque politique possède sa ou ses conditions (les imbrications de conditions sont construites à partir de conditions élémentaires réunies dans une combinaison de conditions `AdaptationPolicyCausesCombination`, afin de simplifier leur prise en compte), ainsi que son ou ses résultats. Il résulte de l'interprétation des politiques une représentation ordonnée des modèles de communication spécifiant leur utilisation : partiellement ou entièrement activé, ou inversement : partiellement ou entièrement désactivé.

La représentation ordonnée des modèles est alors comparée à la représentation "en cours" utilisée par le Manager, et est mise à jour si des modifications

ont eu lieu. Cette représentation est ensuite consultée à chaque communication multi-modèles entreprise par l'infrastructure, et permet au Manager de choisir le ou les modèles à utiliser pour acheminer l'invocation.

Les politiques d'adaptation concernent le choix des modèles pour les communications multi-modèles. Les communications "classiques" avec des composants n'intégrant pas de combinaison de modèles ne sont ainsi pas altérées par le changement de contexte. L'utilisation d'un modèle donné pour transmettre une invocation sur un composant "standard" peut donc avoir lieu même si les politiques engendrent la désactivation de ce même modèle pour une `MultiModelInvocation`.

## *IV.3 Intégration des modèles de communication*

L'implantation de mécanismes pour intégrer des modèles de communication existants à notre infrastructure logicielle multi-modèles pose un certain nombre de problèmes. En effet, dans un premier temps, différents choix d'implantation sont possibles :

- Pour être pleinement intégrés à l'infrastructure et proposer des mécanismes optimaux pour assurer les communications des différents modèles de programmations offerts, les modèles de communications peuvent être redéveloppés intégralement. Ainsi, seules les fonctionnalités réellement utiles à l'infrastructure peuvent être utilisées et adaptées à son fonctionnement interne.
  Si ce premier choix permet de fournir une infrastructure homogène et performante, il nécessite surtout un temps de conception et de développement très important. En outre, développer un modèle de communication déjà existant n'est pas "rentable" : d'autres travaux ont d'ores et déjà été menés par d'autres développeurs afin de fournir des systèmes spécialisés et performants, présentant un nombre important de fonctionnalités abouties. Tenter de développer nos propres modèles conformes aux spécifications risquerait à coup sûr d'aboutir sur une solution moins performante, et surtout moins fiable et complète.
- Si l'on veut éviter de redévelopper un modèle de communication existant, il est possible de choisir de n'utiliser que la personnalité protocolaire d'un intergiciel. Il est alors nécessaire de la découpler de la personnalité applicative de celui-ci.
  Le travail induit par ce choix est conséquent, car les intergiciels sont généralement monolithiques et la séparation des préoccupations n'y est pas évidente. Certains intergiciels modulaires (ou "componentisés")

comme Jonathan [DHDTS98] et OpenORB [BCA+01] ont axé leur conception sur cette séparation ; ils permettent d'obtenir un modèle de communication exempt des autres mécanismes de l'intergiciel. Mais les intergiciels modulaires actuels ne proposent pas toutes les personnalités protocolaires nécessaires à notre infrastructure ; le développement serait donc important pour l'implantation des différents modèles.

– Pour pouvoir utiliser simplement des modèles de communication existants, l'utilisation classique des intergiciels existants offre une solution fiable. Pour chaque utilisation d'un objet réparti, une version complète de l'application est codée pour chaque modèle de communication : e.g. tous les souches et squelettes RMI sont générés, et le choix d'un modèle de communication revient à confier l'invocation entière à l'intergiciel qui le gère.
  Ce choix présente quelques avantages : l'infrastructure profite immédiatement d'intergiciels déjà existants. Cependant, le fait de laisser l'intergiciel gérer de bout en bout les communications empêche l'infrastructure de proposer une combinaison efficace de plusieurs modèles, et l'utilisation du modèle de programmation de l'intergiciel demande d'établir des "wrappers" importants afin d'assurer son utilisation par les différents modèles de programmation de notre modèle générique.
– Enfin, un choix intermédiaire se présente à nous : l'utilisation des intergiciels existants, associée aux mécanismes d'interception et d'introspection offerts par ceux-ci, lorsqu'ils sont disponibles. En effet, l'utilisation d'une partie élémentaire du modèle de programmation de ces intergiciels, ainsi que des intercepteurs et des techniques de type DII et DSI pour la gestion dynamique des communications permet d'obtenir une solution performante et relativement facilement implantable dans l'infrastructure.
  Bien sûr tous les intergiciels ne présentent pas de tels mécanismes, il faudra donc se résoudre à n'utiliser que le modèle de programmation pour certains modèles. Néanmoins, les modèles qui en sont dépourvus sont également ceux qui fournissent un modèle de programmation le plus basique (et donc le moins coûteux à utiliser dans l'infrastructure, e.g. JMS).

### *IV.3.1 Premier intergiciel pris en charge : RMI*

L'infrastructure logicielle multi-modèles que nous avons développée doit s'appuyer sur des modèles de communication connus et reconnus pour leur utilité et leurs performances. C'est pourquoi nous avons choisi d'implanter, comme premier modèle de communication, Java RMI. Effectivement, utilisé

essentiellement pour le développement d'applications J2EE, Java RMI utilise les deux principaux protocoles de communication JRMP et IIOP, il est donc facilement interopérable avec un grand nombre d'applications existantes.

L'intergiciel RMI est intégré à la JVM standard, ce qui signifie qu'aucun programme extérieur n'est nécessaire à l'infrastructure pour adopter ce modèle de communication.

RMI est un intergiciel de communication basé sur un modèle de programmation de type RPC synchrone. Il présente donc des mécanismes adaptés aux invocations bidirectionnelles, l'essentiel du travail d'implantation sera donc concentré autour de l'adaptation de ces mécanismes pour les invocations asynchrones et unidirectionnelles, ainsi qu'autour de l'optimisation de son utilisation, i.e. l'utilisation de tous les moyens techniques offerts par l'intergiciels pour se passer du modèle de programmation et utiliser le plus directement possible le modèle de communication.

*Intégration de l'intergiciel comme modèle de communication dans l'infrastructure*

Java RMI intègre dans la version que nous considérons (Java 5) des mécanismes d'invocation dynamique (DII) permettant de composer une invocation directement à l'exécution, pour utiliser un objet inconnu au moment du développement. Cette possibilité nous permet d'utiliser ce système pour éviter d'une part l'utilisation "classique" du modèle de programmation RMI, qui serait en finalité plus coûteuse à l'exécution, et pour éviter d'autre part le passage par la génération des souches à la compilation.

Ce dernier point est crucial pour l'infrastructure : il permet de conserver la prise en compte des annotations au chargement des classes, l'une des propriétés majeures de notre système.

En ce qui concerne la dynamicité côté serveur (DSI), même si elle n'est pas évidente, l'utilisation conjointe de ASM et RMI la permet toutefois : un objet a besoin d'implémenter une interface descriptive (une sorte de description d'interface IDL, mais en Java) qui hérite elle-même de l'interface `java.rmi.Remote`. Si ce fonctionnement n'autorise pas la dynamicité au premier abord, l'infrastructure peut générer à la volée l'interface de description, et ainsi autoriser la création dynamique d'un objet serveur RMI.

L'utilisation du modèle de communication RMI pour invoquer une méthode sur un objet "purement" RMI ne pose pas de problème particulier, la `MultiModelInvocation` est convertie en appel dynamique RMI sur l'objet distant. L'objet serveur multi-modèles ne pose lui non plus aucun problème particulier, les invocations de méthodes sont directement effectuées sur l'objet "proxy serveur" par RMI.
La communication entre client et serveur multi-modèles est juste une extension de ce principe : l'objet "proxy serveur" présente également une méthode `__MultiModelInvoke`, qui appelle ensuite la méthode cible désignée par l'invocation. En cas d'échec d'envoi de la valeur de retour de l'invocation, l'objet proxy soumet l'invocation retour à son Manager pour qu'il utilise un autre modèle.

**Pour les communications "client"**
Pour transmettre une invocation au "proxy serveur" de l'objet cible, le Manager compose une invocation dynamique RMI à partir de la référence RMI du proxy (souche transmise par le serveur de nommage) et de la déclaration de la méthode ciblée.

Dans le cas d'une invocation multi-modèles, la méthode `__MultiModelInvoke` est ciblée, et l'objet `MultiModelInvocation` est passé comme paramètre. Le même estampillage unique d'invocation est soumis à l'object `MultiModelInvocation` et à l'invocation dynamique RMI. Si une exception est lancée lors de l'invocation dynamique, s'il s'agit d'une invocation RMI l'invocation est alors déléguée à un autre modèle de communication par le Manager, sinon l'exception est propagée ; ce dernier cas ne doit normalement jamais se produire, vu que les exceptions lancées par l'objet serveur sont stockées dans l'objet `MultiModelInvocation`.

Dans le cas d'une invocation classique, la construction de l'invocation est effectuée directement à destination de l'objet serveur, et aucune gestion particulière d'exception n'est mise en place. Seul une imbrication des exceptions RMI est faite dans une exception générique de l'infrastructure, pour assurer l'homogénéité des exceptions gérées par le développeur.

**Pour les communications "serveur"**
Comme nous l'avons précisé plus haut, l'objet "proxy serveur" doit respecter une interface particulière pour pouvoir être accessible via RMI. L'infrastructure génère donc une première interface héritant de `java.rmi.Remote` et présentant les différentes méthodes définies comme accessibles de l'objet

distribué, ainsi qu'une méthode `__MultiModelInvoke` permettant la communication enrichie entre client et serveur multi-modèles.

L'objet "proxy serveur" implémente donc cette interface, et le Manager se charge de l'exporter par RMI, pour le rendre accessible : le paramètre `location` de l'annotation `DistributedObject` désigne l'adresse du serveur de nommage, et `name` désigne le nom symbolique lié à l'objet ; L'objet "proxy serveur" est d'abord exporté, puis déclaré auprès du serveur de nommage.

Le système Java RMI ne permet pas de savoir si la valeur de retour est bien parvenue à l'objet appelant. Nous avons donc dû ajouter une fonctionnalité au "proxy serveur" afin de permettre la réémission de cette valeur par un autre modèle en cas d'échec. Pour cela, lorsqu'une invocation multi-modèles est relayée, la valeur de retour (l'objet `MultiModelInvocation`) est mémorisée dans une table de hachage, l'heure précise y est également associée dans la table. D'autre part l'objet MultiModelInvocation est enrichi pour contenir une liste d'estampilles, correspondant aux identifiants uniques des invocations dont les valeurs de retour n'ont pas encore été reçues par l'objet appelant.

Lorsque l'objet "proxy serveur" reçoit une invocation multi-modèles, il consulte cette liste. Celle-ci lui permet d'effectuer plusieurs opérations :

- Il supprime de sa table toutes les `MultiModelInvocation` provenant du même objet appelant, dont l'estampille est inférieure à celle de l'invocation en cours, et non contenue dans la liste. Clairement, il épure la liste de toutes les valeurs de retour bien reçues par l'objet appelant.
- Il consulte la table et transmet les objets `MultiModelInvocation` destinés à l'objet appelant au Manager pour qu'il utilise un modèle de communication adapté au contexte.
- Enfin, un `Thread` parcourt régulièrement cette table, pour émettre les valeurs de retour (sans distinction d'objet appelant) restantes via le Manager par un modèle adapté au contexte, et également pour supprimer définitivement les valeurs non retournées depuis trop longtemps. Le *timeout* est choisi suffisamment important pour éviter les méprises (déconnexions prolongées mais pas définitives par exemple).

*Gestion des paradigmes de programmation "naturels"*

Une fois le système de référencement de l'infrastructure logicielle multi-modèles adapté pour prendre en compte les références RMI, et le système

RMI enrichi pour fournir également les références des autres modèles, il est nécessaire d'assurer une utilisation efficace du modèle de communication pour tous les modèles de programmation de l'infrastructure. Pour s'assurer de l'efficacité de cette utilisation, il faut réduire au maximum les traitements et communications qui ne sont pas utiles au modèle de programmation.

La gestion des trois types de programmation répartie RPC proposés par notre modèle générique pourrait être similaire, cependant dans ce souci d'efficacité, nous les avons implantés spécifiquement :

- L'invocation bidirectionnelle synchrone est le mode de fonctionnement standard de RMI, il n'y a donc aucun mécanisme particulier à implanter excepté ceux inhérents à la combinaison de modèles. L'objet "proxy client" émet une invocation dans son fil d'exécution, et y restera jusqu'à transmission de la valeur de retour. Dans le cas d'une erreur lors de l'envoi de l'invocation ou de la réception de la valeur de retour, l'objet "proxy client" patiente jusqu'à l'obtention de la valeur par le Manager (ou jusqu'à un temps limite *timeout* prédéfini, et déclenche alors une exception).
- L'invocation bidirectionnelle asynchrone suit le même principe, sauf que le fil d'exécution de l'invocation n'est pas celui de l'objet appelant. La communication RMI de ce type d'invocation reste en tous points identique au précédent : une invocation asynchrone ne l'est effectivement que pour l'appelant ; le modèle de communication et l'objet appelé ont un fonctionnement synchrone.
- L'invocation unidirectionnelle "immédiate" demande, elle, des mécanismes particuliers pour optimiser le fonctionnement de l'infrastructure. En effet, lorsque l'objet "proxy serveur" reçoit l'invocation, il crée un `Thread` chargé d'appeler la méthode ciblée, et retourne tout de suite sans préciser de valeur de retour.
  La prise en charge d'un tel type d'invocation demande de s'assurer que l'invocation a bien été délivrée une et une seule fois, même si l'appelant ne souhaite pas suivre son déroulement.

*Gestion des patrons de conception "émulés"*

Les patrons de conception fournis par le modèle de programmation de l'infrastructure logicielle multi-modèles ne sont pas gérés nativement par RMI, il est donc nécessaire d'utiliser ce modèle de communication simplement mais efficacement pour recréer une communication du même type.

Les files de messages sont des objets répartis d'un type particulier, qui peuvent être physiquement situés à plusieurs endroits en même temps (fonctionnement distribué ou ubiquitaire), comme le permet JORAM [Bal04] par exemple. L'émulation d'une file de messages par RMI ne peut évidemment pas offrir ce type de fonctionnement ubiquitaire, et nécessite de situer physiquement la file à un emplacement précis. La file est donc utilisée simplement comme un objet réparti standard, permettant de produire des messages via une méthode unidirectionnelle, et de les consommer via une méthode bidirectionnelle asynchrone. Cet objet file est défini comme étant l'attribut d'un objet, comme les instances `DistributedObject` des `DistributedClass`.

Les sujets d'intérêt peuvent être considérés quasiment comme des files de messages, à l'exception près que tous les différents objets appelants ayant souscrit au sujet doivent recevoir les mêmes messages. Mais cette nuance concerne l'implémentation de l'objet, et non le modèle de communication. L'utilisation de RMI pour les sujets d'intérêt est donc la même que pour les files de messages.

Les espaces partagés peuvent également être représentés par des objets serveurs, sur lesquels on effectue quelques invocations synchrones : ajout, consultation, suppression. Leur utilisation via RMI ne se différencie donc pas de l'utilisation d'un objet serveur RPC classique.

*Gestion des exceptions et réactions du système*

Comme nous l'avons expliqué plus haut, la gestion des exceptions permet de réagir aux problèmes liés au contexte et d'utiliser la combinaison de modèles de communication en conséquence. Ainsi lorsqu'une exception spécifique à RMI est reçue lors de l'envoi d'une invocation multi-modèles, celle ci est alors soumise au Manager pour être acheminée via un modèle adapté au contexte. Malheureusement, RMI ne permet de savoir si l'exception concerne l'envoi ou la réception de données, et donc si l'invocation a eu lieu ou non. L'infrastructure considèrera donc qu'elle n'a pas eu lieu, et tentera de l'envoyer via un autre modèle. Pour éviter des invocations multiples correspondant à un seul appel, le système de liste de `MultiModelInvocation` décrit plus haut permet de récupérer la valeur de retour de l'invocation si celle-ci a déjà eu lieu, et d'éviter ainsi des doublons.

Bien sûr, si l'invocation provoque une exception prévue par l'objet serveur, celle-ci sera alors propagée jusqu'à l'appelant.

### *IV.3.2 Intergiciel de type asynchrone : JMS*

L'intérêt de notre infrastructure logicielle multi-modèles n'est réel que si les modèles de communication combinés sont bel et bien complémentaires. C'est pourquoi nous avons implanté comme second modèle JMS, un intergiciel asynchrone permettant de communiquer via des files de messages et des sujets d'intérêt.

La combinaison des deux modèles implantés est alors bénéfique : les deux modèles de communication ne sont pas adaptés aux mêmes situations, ne présentent pas les mêmes performances ni les mêmes mécanismes internes. Notre approche est ainsi parfaitement illustrée.

Plusieurs implémentation des spécifications JMS existent aujourd'hui, certaines commerciales, d'autres opensource. A l'instar de RMI pour le RPC, JMS est le mode de communication orienté messages le plus utilisé dans J2EE, au travers de serveurs d'applications tels que WebSphere. Nous avons choisi d'utiliser l'implémentation JORAM pour implanter JMS dans l'infrastructure multi-modèles. D'une part il s'agit d'un intergiciel opensource, performant et léger, et d'autre part il présente l'avantage de ne dépendre d'aucun serveur d'application ni d'aucune autre technologie (il peut toutefois très bien s'intégrer à un serveur d'application).
Nous avions déjà choisi cette implémentation pour le développement du greffon multi-modèles, et ce choix s'est révélé judicieux et fort satisfaisant, il nous a donc semblé logique de continuer à l'utiliser.

L'utilisation d'un intergiciel orienté messages (MOM) induit certaines limitations et obligations. Ainsi, les messages transitent par un (ou plusieurs) serveur intermédiaire, qui peut être au niveau du producteur, du consommateur, ou bien à un autre emplacement indépendant. Les différents serveurs sont capables de communiquer entre eux, de se synchroniser, et de répartir leur travail afin de présenter un comportement homogène.

Pour implanter JMS à l'infrastructure, le choix a donc été possible entre une situation où le serveur est indépendant et un système où chaque composant réparti possède son propre serveur. Dans le premier cas, la problématique principale est de gérer ce serveur. En effet, étant indépendant de l'infrastructure, elle n'a aucun contrôle (ou presque) dessus. Nous avons donc opté pour le deuxième cas, en intégrant un serveur à chaque Manager. Bien que cette solution soit coûteuse, elle permet une meilleure gestion du système de messages. Ce choix n'est bien sûr pas figé, et si d'autres solutions

plus performantes étaient démontrées, nous pourrions rapidement changer le déploiement du système.

*Intégration de l'intergiciel comme modèle de communication dans l'infrastructure*

Comme nous l'avons dit plus haut, le but de l'implantation des intergiciels est d'obtenir une intégration efficace et performante, en optimisant l'utilisation de ceux-ci tant que possible.

Si le modèle de programmation de RMI présentait des couches qui ne nous étaient pas utiles, celui de JMS est beaucoup moins étendu. En fait il consiste en une composition explicite des messages, se rapprochant de l'interface d'invocation dynamique (DII) offerte par RMI. Les objets proxy utilisent donc cette composition de message pour acheminer l'invocation et son retour, d'une façon similaire à l'utilisation du modèle synchrone.

*Gestion des patrons de conception "naturels"*

Les files de messages ainsi que les sujets d'intérêt sont les modèles naturellement fournis par JMS. Leur utilisation lors de communications mono-modèle est donc élémentaire : une file de messages ou un sujet du modèle de programmation correspond au même concept au niveau du serveur JORAM.

Le fonctionnement lors de communications multi-modèles diffère cependant : comme nous l'avons précisé au sujet de l'implantation de RMI, les patrons de conception fournis par le modèle de programmation générique sont représentés par des objets RPC. Les messages envoyés sont donc en finalité interprétés comme des invocations ; ce système est inévitable pour permettre l'utilisation conjointe de modèles de communication synchrones et asynchrones pour utiliser ces patrons de conception.

*Gestion des patrons et paradigmes "émulés"*

L'utilisation de JMS pour acheminer des invocations de type RPC nécessite des mécanismes d'adaptation particuliers. Ainsi lorsque l'invocation est multi-modèles, le Manager envoie un message de type `ObjectMessage` contenant la `MultiModelInvocation`.

Lorsque l'invocation est mono-modèle, deux cas sont possibles :

- Le client n'est pas multi-modèles, il peut alors utiliser l'objet serveur RPC en envoyant dans la file de messages correspondante un message contenant les paramètres spécifiques à la méthode appelée. L'objet "proxy serveur" effectue donc l'invocation à réception du message, et renvoie un message contenant la valeur de retour si l'appel est bidirectionnel.
- Le client multi-modèles effectue une invocation RPC destinée à produire un message dans une file ou un sujet d'intérêt. Dans ce cas l'objet "proxy client" est un patron de conception qui fournit les méthodes correspondantes au fonctionnement de JMS, l'adaptation est donc minimale.

Pour presque toutes les invocations de type RPC, l'utilisation de JMS oblige l'appelant à désigner le chemin par lequel la valeur de retour devra passer ; la `MultiModelInvocation` contient donc la référence de la file de messages utilisée par l'appelant pour recevoir le retour de l'invocation.

Si l'implantation d'un tel intergiciel n'a pas soulevé beaucoup de problèmes ni de verrous, son utilisation dans le cadre d'une invocation multi-modèles se limite à une file de messages contenant un objet. Même si les bénéfices d'une implantation plus complexe ne semblent pas évidents, l'utilisation d'autres types de messages, ou de sujets d'intérêt pourrait sans doute optimiser les communications dans certains cas.

*Gestion des exceptions et réactions du système*

JMS propose un système exempt d'exceptions inopportunes dans les situations basiques, dû à l'utilisation de plusieurs serveurs et d'un modèle de programmation asynchrone. E.g. lorsqu'un message ne parvient pas à être délivré, il reste simplement dans la file de messages en attendant une reconnexion. Les exceptions que l'on peut rencontrer sont alors des problèmes de droits d'authentification ou de file/sujet inexistant. La seule exception exploitable par l'infrastructure est une `JMSException` déclenchée lorsque le serveur ne peut être joint. Le serveur étant local dans notre implémentation, si ce cas se présente, un problème plus grave risque d'en être la cause (e.g. erreur générale de JORAM).

Les exceptions déclenchées par la méthode invoquée n'ont pas lieu d'être gérées par JMS. En effet, dans les trois cas d'utilisation, si le client est mono-modèle il n'a aucun moyen de gérer cette exception, si le serveur est mono-

modèle il ne peut pas déclencher d'exception via JMS, et lors d'une `MultiModelInvocation` l'exception est déjà encapsulée dans l'objet.

### *IV.3.3 Perspective de prise en charge future : CORBA*

Le bien fondé de l'infrastructure logicielle multi-modèles a pu être justifié par l'implantation de deux modèles de communication complémentaires, mais l'un des objectifs de cette infrastructure est d'assurer l'interopérabilité avec des composants répartis utilisant des modèles de communication variés. L'implantation de CORBA comme modèle supplémentaire permettrait une ouverture supplémentaire sur des composants plus hétérogènes ; CORBA permet en effet de faire communiquer des objets et composants développés sous plusieurs langages, sur plusieurs plateformes d'exécution.

CORBA propose différents modèles de programmation, comme le RPC bidirectionnel et unidirectionnel, ainsi que les événements. Il propose ainsi des mécanismes efficaces de gestion de ces modèles au travers de son modèle de communication.

Les mécanismes qui permettraient d'implanter CORBA dans l'infrastructure sont nombreux et efficaces : outre le modèle de programmation qui permet, comme RMI, de rendre implicite l'utilisation du modèle de communication, il propose des interfaces d'invocation dynamique (DII), ainsi que de serveur dynamique (DSI), et également un système d'intercepteurs capables d'observer et d'intervenir à différents stades des invocations. Ce dernier système permettrait d'optimiser les communications multi-modèles en informant directement l'objet "proxy serveur" du succès ou non de l'envoi de la valeur de retour.

#### *Modèles de programmation multiples*

La diversité des modèles de programmation présentés par CORBA est bien sûr un atout. En effet, plus l'intergiciel implanté propose de modèles, et moins de mécanismes d'adaptation devront être implémentés pour permettre l'utilisation efficace du modèle de communication par l'infrastructure.

La particularité de CORBA est son langage de description d'interfaces : IDL. Si de telles interfaces ne permettent pas forcément de simplifier l'adaptation dynamique, elles permettent néanmoins d'uniformiser les objets répartis en assurant que la souche et le squelette d'un objet serveur respectent la même interface.

*Discussion sur l'intérêt de cette prise en charge*

L'implantation de CORBA pourrait compléter notre infrastructure en y apportant une grande interopérabilité et une gestion ainsi qu'une surveillance efficace des communications.

Néanmoins, le modèle de communication proposé par CORBA peut également être utilisé par RMI, ce qui assure le même degré d'interopérabilité que CORBA. Seuls restent les mécanismes évolués qu'il fournit. Ils seraient sans doute performants, mais l'évolution de l'infrastructure serait plus de l'ordre de l'optimisation que de l'enrichissement des fonctionnalités. L'ajout de CORBA à l'infrastructure n'est donc pas un complément d'intérêt pour notre approche, mais juste une amélioration technique de notre implémentation.

## *IV.4 Version statique de l'infrastructure multi-modèles*

Pour présenter une infrastructure logicielle multi-modèles simple et performante, nous avons choisi dans un premier temps de développer une version de celle-ci qui n'intègre pas tous les mécanismes d'adaptation dynamique présentés précédemment.

Cette première version de l'infrastructure se base donc sur un compilateur intermédiaire, prenant en compte les annotations présentes dans le code pour générer tout le code nécessaire à l'exécution multi-modèles de l'application. L'intérêt de cette démarche est de fournir une application compilée définitivement, sans qu'elle ne fasse appel à des techniques de modification de bytecode durant l'exécution. L'application obtenue présente donc un comportement figé, correspondant aux politiques d'adaptation définies lors de la compilation intermédiaire. L'absence de changement du code de l'application pendant qu'elle s'exécute lui permet d'économiser des ressources (temps de calcul et mémoire). Ce type d'optimisation peut être appréciable pour un déploiement sur des plateformes à ressources limités (e.g. systèmes embarqués ou mobiles).

Toutefois, cette version "statique" de l'infrastructure multi-modèles présente évidemment des limitations dûes à ce code immuable :

- Tout d'abord, aucun changement de politiques d'adaptation n'est envisageable au cours de l'exécution. Le fichier de politiques est interprété lors de la phase de compilation intermédiaire et est ainsi appliqué

définitivement au mécanismes internes de l'application. Ceci signifie par conséquent que l'utilisation de politiques différentes en fonction du contexte de déploiement de l'application est à proscrire ; une compilation intermédiaire distincte pour chaque contexte de déploiement est nécessaire, on obtiendra donc autant de versions de l'applications que de fichiers de politiques d'adaptation distincts.
– L'architecture multi-modèles comme nous l'avons décrite précédemment permet à une classe d'être modifiée (y compris et surtout au niveau de ses annotations) puis redéfinie, l'agent de supervision la prend en charge automatiquement et regénère les mécanismes appropriés. Dans la version présente, ce type de comportement est impossible. L'aspect réflexif de l'application est donc inenvisageable.
– Le chargement de nouvelles classes annotées non connues lors de la compilation intermédiaire n'est pas géré par cette version (les annotations seraient tout simplement ignorées). L'application doit donc nécessairement définir une fois pour toutes sa répartition avant la phase de compilation multi-modèles, sous peine de ne pas intégrer les mécanismes nécessaires.

Néanmoins, l'infrastructure garde dans cette version tous ses aspect d'adaptation au contexte, de programmation générique et de communications multi-modèles. Seuls les aspects d'adaptation dynamique sont écartés.

### *IV.4.1 Le Manager devient un compilateur offline*

Le Manager dans l'infrastructure multi-modèles que nous avons présentée précédemment occupe le rôle d'interprète des politiques d'adaptations, ainsi que celui de gestionnaire des communications multi-modèles. C'est essentiellement lui (avec que l'agent de chargement de classes) qui présente les mécanismes dynamiques de l'infrastructure. Ces mécanismes étant absents dans cette version, le rôle du Manager se simplifie, et sa place dans l'infrastructure exécutée diminue.

Mais les mécanismes dynamiques font place à un système de génération définitive du code d'adaptation, qui peut intervenir avant le déploiement, par conséquent avant l'exécution. Les mécanismes sont générés lors de l'étape de compilation intermédiaire de l'application : après la compilation "traditionnelle", l'application annotée est prise en charge par le compilateur multi-modèles. Ce dernier intègre alors tous les mécanismes normalement générés à la volée lors de l'exécution de l'infrastructure à l'application compilée. Celle-ci peut alors être exécutée sans l'aide d'aucun paquetage additionnel (autre que

les intergiciels utilisés) : elle gère "seule" (i.e. sans programme superviseur additionnel) ses communications.

A partir des annotations, le compilateur intermédiaire génère les classes "proxy client" chargées de gérer les références vers les objets serveur, ainsi que les classes "proxy serveur" comme définies dans la section IV.2.2. Les différents types d'invocation sont définis statiquement, ainsi l'objet "proxy client" utilisé exclusivement en RPC synchrone bidirectionnel n'intègre pas les méthodes et mécanismes pour être utilisé via un autre modèle de programmation.

### *IV.4.2 Optimisations et simplifications des mécanismes d'adaptation*

Le compilateur interprète les politiques d'adaptation afin de créer un générateur de représentations ordonnées des modèles de communication : à partir de ces politiques, une classe `AdaptationPolicyInterpreter` est générée, chargée de fournir au Manager la représentation de modèles correspondante au contexte courant. Si cette classe construit dynamiquement ses mécanismes dans l'infrastructure complète, elle les définit une fois pour toutes dans cette version "statique". Nous aurions pu implémenter la création d'une classe contenant toutes les représentations en fonction de toutes les observations de contexte possibles, se rapprochant du principe de fonctionnement de Qinna [TBO05]. Cependant, ce type de conception peut vite devenir coûteux en espace mémoire, car le risque d'explosion combinatoire des "éventualités de représentations" est certain.

Les observateurs de contexte restent inchangés dans cette version de l'infrastructure. En effet ils n'intègrent en aucun cas de mécanismes dynamiques, et leur gestion par le Manager n'a pas de raison particulière d'être différente dans cette version. Seul le fonctionnement de la génération des représentations en fonction de leurs observations devient statique, ce qui ne modifie en rien l'utilisation de ces représentations par le Manager.

Que l'infrastructure logicielle multi-modèles présente ou non des mécanismes d'adaptation dynamique, son fonctionnement global reste identique, et elle représente toujours notre approche multi-modèles tant au niveau de son modèle de programmation que de sa combinaison de modèles de communication.
Les deux versions de l'infrastructure sont évidemment compatibles, i.e. un client statique et un serveur dynamique peuvent communiquer sans que cela

induise un quelconque changement dans leur fonctionnement. Il est clair que chaque version s'adapte à un contexte de déploiement particulier, la version statique offre une application "finie" (i.e. qui ne sera plus modifiée) présentant une utilisation des ressources plus optimisée ; son exécution est adaptée à des plateformes présentant des ressources limitées.
La version dynamique (cf section suivante) offre plus de fonctionnalités dynamiques, plus coûteuses, mais permet ainsi un déploiement plus optimisé, i.e. les politiques peuvent être modifiées pour modifier le fonctionnement multi-modèles de l'application, sans devoir modifier l'application elle-même.

## *IV.5 Infrastructure logicielle multi-modèles : version fonctionnelle*

L'infrastructure logicielle multi-modèles comme nous la présentons dans ce chapitre intègre des mécanismes dynamiques capables de reconfigurer dynamiquement le fonctionnement multi-modèles de l'application. C'est la version fonctionnelle que nous avons développée en finalité de nos travaux.

### *IV.5.1 Prise en compte plus dynamique du modèle de programmation*

L'avantage premier de cette version de l'infrastructure est l'absence de compilateur intermédiaire : l'application est compilée originalement en conservant ses annotations, et c'est seulement au moment du chargement des classes par la JVM qu'un agent de supervision inspecte les classes compilées afin d'y chercher les annotations multi-modèles.

Si la modification des classes annotées devient dynamique, par conséquent les objets générés en complément de ces modifications le sont également. Ainsi les objets "proxy client" et "proxy serveur" relatifs à une invocation sont créés à la volée lors du chargement de la classe dont ils dépendent.

L'utilisation de l'application devient dans cette version moins restreinte : les classes compilées ne contiennent que des annotations, elles peuvent donc être utilisées sans l'infrastructure pour présenter un comportement local (i.e. non distribué). Par exemple, une application de gestion de stock est développée pour être utilisée en local sur un poste : elle est parfaitement autonome. Pour permettre une utilisation distante de cette application, le développeur ajoute des annotations définissant l'objet au cœur du code métier comme utilisable par d'autres applications distantes. L'application ainsi compilée peut

donc soit être exécutée avec l'infrastructure – l'objet annoté sera alors accessible à distance –, soit exécutée seule – les annotations n'auront aucune incidence sur son fonctionnement –.

La prise en charge à la volée des annotations permet également de rendre l'application répartie extensible : des classes peuvent être ajoutées en cours d'exécution (chargement local ou distant, par `URLClassLoader` ou via un système comme OSGi), et leurs annotations seront prises en compte dès leur chargement pour intégrer les mécanismes multi-modèles.

### *IV.5.2 Réévaluation des politiques en cours d'exécution*

En complément du chargement de classes annotées à la volée, les politiques d'adaptation sont surveillées dans cette version. En effet, le fichier de description des politiques est susceptible d'être modifié au cours de l'exécution de l'application, et le Manager vérifie régulièrement si des changements sont apparus dans ce fichier. Si tel est le cas, une nouvelle représentation ordonnée des modèles de communication utilisables est construite, puis comparée à la représentation actuellement utilisée, qui sera remplacée si des changements sont apparus.

Ce système de prise en compte des modifications du fichier de politiques offre à l'infrastructure, et donc à l'application qui l'utilise, des possibilités de reconfiguration dynamique que ne permettait pas la précédente version de l'infrastructure. Cette reconfiguration peut être appréciable lors de changement de déploiement ou lorsqu'une autre application, ayant une perception différente du contexte, veut orienter les politiques pour optimiser l'adaptation à ce contexte.

## *IV.6 Synthèse et conclusion sur les implémentations*

Nous avons proposé différentes implémentations de notre approche afin d'illustrer et de valider nos travaux. Elles implantent chacune certains des éléments caractéristiques de notre approche, et montrent et justifient ainsi les bénéfices possibles de leur utilisation.

Dans un premier temps, nous avons développé un greffon multi-modèles pour appliquer l'approche multi-modèles à un projet concret proposant un accès Internet à bord des trains. La problématique principale du projet réside dans les déconnexions de la liaison satellitaire, et notre approche a fourni

une solution à celle-ci. Nous avons développé dans ce sens un greffon multi-modèles, capable d'enrichir le système de communication de n'importe quelle application pour permettre l'utilisation combinée des `Socket` et de JMS. Le système résultant propose l'utilisation transparente et dynamique du modèle de communication le plus adapté aux conditions courantes. Une évaluation de ce greffon a permis de comparer ses performances face à un système classique, et le surcoût minime (prévisible du fait des mécanismes ajoutés au système d'origine) en termes de temps de réponse n'est pas significatif et ne contredit en rien l'intérêt certain de cette proposition, de notre approche ainsi que de l'infrastructure.

Dans un deuxième temps, nous avons conçu et développé les éléments et mécanismes clés de l'infrastructure logicielle multi-modèles :

- Le modèle de programmation générique fournit des annotations permettant une programmation simple et rapide d'applications réparties, communiquant via des mécanismes synchrones, asynchrones et unidirectionnels. De plus, des patrons de conception fournissent des outils de communication recréant des principes comme les files de messages, les sujets d'intérêt et les espaces partagés. Enfin, des méthodes explicites ajoutent des possibilités de personnalisation et de dynamicité des communications dans l'application.
- Le chargement dynamique des classes annotées de l'application, géré par un agent de supervision associé à l'environnement d'exécution Java, permet à celles-ci de rester inchangées jusqu'à l'exécution de l'application, et rend possible l'ajout dynamique à l'exécution de nouvelles classes annotées inconnues à la compilation. Lors du chargement des classes, les classes proxy sont également générées à la volée, offrant des possibilités de mise à jour de l'infrastructure plus aisées : si l'infrastructure évolue et change ses mécanismes internes, l'application n'a aucun besoin d'être recompilée ou redéveloppée, c'est l'infrastructure elle-même qui génèrera les nouveaux mécanismes lors du chargement : l'application annotée n'est pas dépendante de la version de l'infrastructure utilisée lors du développement ni de la compilation.
- Les observateurs de contexte soumettent leurs observations au Manager de l'infrastructure, pour pouvoir guider l'interprétation des politiques afin d'obtenir une combinaison de modèles de communication adaptée au contexte. Ces observateurs peuvent présenter un fonctionnement actif ou passif, afin de produire des observations sous forme de données chiffrées ou booléennes. Des observateurs basiques ont été développés, mais le développeur peut facilement fournir ses propres observateurs

adaptés au domaine et au contexte de développement ainsi qu'aux cas d'utilisation de l'application.
– La gestion des politiques d'adaptation en temps réel permet de s'abstraire complètement de celles-ci lors du développement de l'application. En effet, les mécanismes de sélection des modèles de communication en fonction du contexte et des politiques sont construits à l'exécution, lors de la lecture et de l'interprétation des politiques. En plus d'autoriser l'établissement des politiques d'adaptation après la compilation de l'application, cette gestion dynamique peut également prendre en compte les modifications effectuées sur le fichier de politiques au cours de l'exécution, permettant une reconfiguration dynamique du fonctionnement multi-modèles de l'infrastructure.

Nous avons ensuite implanté différents intergiciels de communication pour pouvoir profiter de leur modèle de communication. L'utilisation de ces intergiciels a nécessité des mécanismes d'adaptation pour que le modèle de programmation générique de l'infrastructure puisse utiliser efficacement chaque modèle de communication, que ce soit en situation d'interopérabilité (i.e. communication avec une application utilisant l'intergiciel original) ou lors de communication multi-modèles enrichies (i.e. tous les composants de l'application présentent des fonctionnalités multi-modèles). Le modèle RPC synchrone proposé par RMI a donc été adapté à l'infrastructure, permettant des communications via les protocoles JRMP et IIOP, puis le modèle orienté messages proposé par JMS. La perspective d'ajout du modèle proposé par CORBA a été discutée, mais les besoins de l'infrastructure ainsi que les bénéfices que présenterait cet ajout ne le justifient pas pour le moment.

Enfin, nous avons développé deux versions de l'infrastructure logicielle multi-modèles : une première version "statique", basée sur une re-compilation de l'application avant l'exécution, puis une seconde version, entièrement dynamique. La version "statique" n'intègre pas le chargement dynamique des classes annotées, ni la gestion en temps réel des politiques d'adaptation. Si certains avantages de l'infrastructure complète ne sont donc plus présents, elle offre toutefois les mêmes capacités d'interopérabilité et d'adaptation aux observations de contexte.
La version "complète" de l'infrastructure intégre tous les concepts énoncés, et permet ainsi d'ajouter le chargement (et la modification potentielle) des classes, et prend en compte les politiques seulement à l'exécution, en autorisant leur modification à la volée.

L'infrastructure logicielle multi-modèles que nous avons développée offre un certain nombre de fonctionnalités, justifiant de l'intérêt de nos travaux : modèle de programmation complet (générique) et clair (annotations), gestion dynamique des changements de contexte (observateurs), de règles d'adaptation (politiques), des modifications et ajouts de l'application (chargement dynamique), interopérabilité avec les intergiciels existants, et amélioration des communications grâce au principe multi-modèles et à la combinaison dirigée par les politiques d'adaptation.
Mais bien sûr l'infrastructure dans son état actuel peut encore subir beaucoup d'améliorations, celles-ci permettraient d'une part de mieux justifier encore l'intérêt de notre approche, et d'autre part d'optimiser les performances et le fonctionnement général de l'infrastructure :

- Le système d'optimisation de code d'appel ne gère que l'utilisation dans un objet de l'un de ses attributs annotés. On peut envisager dans un premier temps de permettre une optimisation dans des cas moins "statiques" : le système d'annotations Java le permettant, la déclaration d'une variable locale annotée pourrait également conduire à une optimisation.
- L'infrastructure a été conçue pour optimiser les communication distantes, mais aucun mécanisme n'a encore été implémenté pour gérer plus efficacement les communications locales : e.g. si la politique correspondant au contexte déclare la désactivation du modèle synchrone (e.g. RMI), toutes les communications seront assurées par les autres modèles, y compris en local, alors que dans ce cas précis le modèle synchrone serait tout de même efficace, et éviterait l'utilisation inutile et couteuse d'un serveur intermédiaire. Une optimisation permettant de gérer ce type de situation permettrait d'améliorer les performances de notre infrastructure.
- Les patrons de conception ne gèrent pas non plus pour le moment les communications locales. Une amélioration pourrait consister à détecter une utilisation locale, et ainsi appeler directement le proxy client ou serveur, plutôt que de solliciter un modèle de communication.
- L'utilisation en partie de l'intergiciel RMI, comme nous l'avons expliqué en section IV.3.1, n'est pas aussi optimisée que nous aurions pu l'espérer, du fait de l'absence de mécanismes de type "intercepteur". On peut songer à utiliser alors le résultat de travaux dans ce domaine[1] pour permettre à l'infrastructure de connaître plus précisément l'état des communication par ce modèle de communication.

[1] Projets visant à fournir des intercepteurs pour RMI, notamment `http ://sourceforge.net/projects/wenbozhu/`

– La composition de l'application distribuée via notre infrastructure dépend actuellement des services de nommage des différents intergiciels utilisés. Une extension possible du système pourrait proposer des mécanismes de type `ObjectToString` appliqués aux objets proxy, afin d'être en mesure de sauver/restaurer ces objets, et d'envisager une évolution fournissant un service de persistance.

Ces améliorations possibles ne remettent en aucune façon en cause le bien fondé ainsi que l'intérêt de l'infrastructure logicielle multi-modèles dans son état actuel, mais permet d'envisager encore de nouvelles possibilités d'utilisation, justifiant d'autant plus les bénéfices potentiels de l'utilisation de l'infrastructure.

# V

# Impact des travaux

Les travaux entrepris et menés durant cette thèse l'ont été dans un contexte bien particulier. Tout d'abord, la thèse a été financée par l'INRETS et la région Nord-Pas de Calais, et même si elle traite d'un sujet informatique (et plus particulièrement applications réparties), c'est son domaine d'application qui a dirigé sa diffusion et sa promotion : les transports ont en effet un intérêt certain dans ce type de projet. Les communications embarquées souffrent fréquemment des conséquences de la mobilité, et la conception d'applications spécifiques au domaine transport se doit de prendre en compte ces inconvénients. Un certain nombre de valorisations de ces travaux sont donc en rapport direct avec les transports.

## *V.1 Implication dans un projet : REVE*

### *V.1.1 Similitudes avec les travaux de thèse*

REVE [JCLC05] est un projet financé par l'ANR dans le cadre des Actions de Recherche Amont Sécurité, Systèmes embarqués et Intelligence Ambiante. Le but du projet est de construire un modèle de composant, un support d'exécution et un système de typage pour les applications embarquées du domaine du temps-réel dans lequel des politiques de prise en compte des changements de contexte peuvent être spécifiées, programmées et vérifiées. Il est mené conjointement par plusieurs partenaires de différentes institutions : l'INRIA Futurs (projet ADAM), le CEA, l'équipe CEDRIC du CNAM, le CITI de l'INSA, et l'INRETS.

Les travaux concernant l'infrastructure logicielle multi-modèles ont également été menés dans le cadre de ce projet. De grandes similitudes demeurent entre les préoccupations du projet et de la thèse. C'est pourquoi nous nous sommes joints à ce projet.

Dans les différents buts que se fixe le projet, nous avons travaillé plus particulièrement sur l'établissement d'un modèle de communication adaptable, ce qui rejoint tout à fait les travaux de la thèse. Nous travaillons également sur une étude de cas et la conception d'un démonstrateur, en lien avec notre domaine d'application, mais ceci est plus éloigné de notre sujet.

Le modèle de communication décrit dans ce projet ne traite aucunement d'interopérabilité, nous avons pu donc optimiser notre approche pour proposer un système plus performant (entre autres, le système de nommage peut être surchargé pour fournir des références enrichies via chaque modèle de communication, ce qui n'était pas envisageable tant que la compatibilité avec les intergiciels classiques devaient être conservée).

#### *V.1.2 Livrables*

Dans le cadre du projet REVE, nous avons participé activement en produisant des livrables. Tout d'abord, une étude de l'état de l'art des modèles de communication [BG06], et plus particulièrement les intergiciels adaptables, afin d'établir l'existant et les besoins dans ce domaine pour le projet. Ce document est la base de nos travaux dans ce projet.

Pour assurer notre rôle au sein du projet, nous avons également travaillé sur un livrable proposant un modèle adaptable de communication pour modèle de composant [GB08], basé sur les travaux de la thèse.

### *V.2 Publications et communications*

Lors de ces trois années de recherche, nous avons cherché à valider notre approche et nos travaux d'application de cette approche au travers de publications et présentations dans la communauté scientifique. Les différents retours, observations et avis que nous avons pu recevoir nous ont été profitables. Ils nous ont permis de remettre en question certains points de notre approche, et de nous conforter dans l'intérêt qu'elle représente.

Ne figurent ici que les "traces écrites" de notre impact, nous avons toutefois également présenté nos travaux lors de réunions et séminaires d'équipe de recherche et de laboratoire, ce qui nous a également été bénéfique : les retours que nous avons eus, moins formels et plus directs, ont été d'une grande utilité dans la poursuite de nos travaux.

### *V.2.1 UbiMob 2006*

Les journées francophones UbiMob traitent de la mobilité et de l'ubiquité. Elles sont issues du groupe de travail GtMob (Informatique Mobile, Ubiquitaire et Evanescent) du GdR $I^3$ (Informatique, Interaction, Intelligent). Les communications dans le domaine mobilité / ubiquité sont évidemment primordiales, c'est pourquoi nous avons tenu à valider notre approche auprès de cette communauté.

L'article soumis à ces journées [BGG06] présente notre approche multi-modèles, ainsi que les bénéfices espérés en application à la mobilité. Bien que l'architecture exacte de notre infrastructure logicielle multi-modèles n'était alors pas définie précisément, les principes généraux étaient déjà établis et permettaient de percevoir le réel intérêt de l'approche.

### *V.2.2 Journées des doctorants SPI INRETS 2006 et 2007*

Le LEOST organise chaque année une journée dédiée aux doctorants des différents laboratoires de l'INRETS. Elle permet à ces jeunes chercheurs d'exposer leurs travaux à un ensemble hétérogène de chercheurs, établis ou autres doctorants, de domaines variés, mais ayant en commun le domaine d'application des transports.

Les articles rédigés pour l'occasion sur deux années ont permis dans un premier temps de présenter clairement notre approche multi-modèles [Boc06], et ses applications possibles aux transports. L'hétérogénéité de l'auditoire lors de la présentation a permis d'obtenir des remarques élémentaires qui n'auraient pas eu lieu d'être face à un public spécialisé. Ce type de retour est très bénéfique, et nous oblige à reconsidérer les fondements mêmes de nos travaux.

Dans un deuxième temps, nous avons présenté plus précisément notre infrastructure logicielle multi-modèles [Boc07], en tâchant de mettre en valeur les bénéfices offerts au domaine des transports. Ici encore les observations de

non spécialistes du domaine ont été au moins aussi profitables que si elles provenaient d'un chercheur familier des principes présentés.

### *V.2.3 ITS 2007*

La conférence européenne ITS traite des systèmes de transport intelligents sous toutes leurs formes (transports guidés, collectifs ou individuels, de personnes ou de marchandises). Nous avons tenu à publier un article dans cette thématique afin de présenter l'application de nos travaux aux transports en général. L'article que nous avons soumis [BG07a] présente notre greffon multi-modèles et son application au proxy Internet embarqué. Cet exemple concret d'application de notre approche multi-modèles démontre l'intérêt certain des mécanismes d'adaptation et de la prise en compte du contexte pour optimiser et sécuriser les communications en mobilité. La présentation de nos travaux avec un discours "grand public" nous a permis de prendre du recul sur ces travaux.

### *V.2.4 ITS-T 2007*

ITS-T est une conférence internationale présentant les innovations de la recherche en matière de télécommunications appliquées aux systèmes de transports intelligents. Comme pour la conférence ITS, l'article [BG07b] que nous avons soumis à ITS-T est voué à présenter l'intérêt de notre approche pour son application aux transports. Son contenu a été plus axé sur l'infrastructure elle-même, et les innovations qu'elle intègre. ITS-T rassemble des spécialistes en réseau et télécommunications, et confronter notre approche aux besoins et problèmes de cette communauté a permis de recadrer les objectifs de nos recherches.

### *V.2.5 EuroDoc Info 2008*

L'école doctorale Sciences pour l'Ingénieur de l'université de Lille organise maintenant chaque année des journées consacrées aux travaux des doctorants en informatique. J'ai pu au travers de ces journées présenter nos travaux auprès de jeunes chercheurs de domaines connexes [Boc08]. Les retours ainsi obtenus ont eu l'avantage de provenir de personnes variées, aux compétences et domaines de recherche éclectiques.

### *V.2.6 SpaceAppli 2008*

Dans le cadre d'un salon sur les technologies satellitaires et leurs applications à Toulouse, des conférences ont été organisées. Nous y avons présenté

le travail effectué sur le greffon multi-modèles [BG08a]. Si ce sujet a déjà été présenté auparavant, son intérêt a ici été plus justifié que précédemment. En effet les projets commerciaux d'accès Internet à bord des trains ont été également présentés, et le constat fut clair qu'aucune solution réellement adaptée aux problèmes de déconnexions satellitaires n'est prévue. Ce système de greffon est donc toujours d'actualité et comble encore aujourd'hui une lacune non gérée.

#### *V.2.7 WCRR 2008*

Conférence mondiale sur la recherche ferroviaire, WCRR a réuni à Séoul les principaux acteurs mondiaux des transports de biens et de personnes par ferroutage. Le système de notre greffon ayant été bien publié, nous avons tenu à soumettre un article exposant l'infrastructure logicielle multi-modèles dans son aboutissement au terme de cette thèse [BG08b]. Des représentants et chercheurs des opérateurs ferroviaires se sont intéressés à notre proposition, qui permet, au delà d'un simple proxy adapté, de concevoir des applications communicantes simplement, et de leur assurer un déploiement optimal et une exécution adaptée à leur contexte.

### *V.3 Synthèse de l'impact de nos travaux*

Par les articles, présentations et discussions engagées dans les conférences, projets et séminaires, nous avons exposé notre approche, son application concrète et notre infrastructure logicielle multi-modèles à différentes communautés de chercheurs. L'intérêt d'une telle approche, ainsi que les bénéfices offerts par un système adaptable ont intéressé, voire convaincu.

Les confrontations avec des points de vue opposés ou divergents ont permis de remettre en question notre approche, et d'étayer notre argumentaire et notre raisonnement.

La participation à un projet tel que REVE nous a de plus permis de confronter notre approche à une problématique différente. Ainsi nous avons pu envisager d'autres applications à nos travaux que l'infrastructure logicielle multi-modèles comme nous l'avons conçue.

# VI

# Conclusion

## *VI.1 Intergiciels actuels et mobilité*

Un grand nombre d'intergiciels existent aujourd'hui, et si tous proposent un fonctionnement général semblable, chacun possède ses spécificités. Tout d'abord, chaque intergiciel présente un modèle de programmation : invocations synchrones, asynchrones, ou bien communications particulières comme des files de messages. Chaque modèle de programmation est adapté à un type de développement particulier.
Ensuite, chaque intergiciel présente un ou plusieurs modèles de communication, adaptés à différents contextes de déploiement.

L'état de l'art tend à démontrer que les modèles de programmation ne proposant qu'un type de communications ne sont pas adaptés à tous les développements. Un modèle "générique", intégrant plusieurs types de communications, serait plus à même de fournir un outil efficace d'abstraction des communications au développeurs.

La mobilité induit une diversité des composants répartis utilisés par une application. Cette diversité se situe d'une part au niveau du type de composants, et d'autre part au niveau de l'intergiciel utilisé par ceux-ci pour être accessibles (ou pour accéder à d'autres composants). Cette hétérogénéité nécessite inévitablement l'utilisation simultanée de plusieurs modèles

de communication, pour permettre l'interopérabilité avec les différents composants.

De plus, cette mobilité induit également des changements de contexte. Pour qu'une application soit efficacement déployée en mobilité, il faut lui fournir des mécanismes d'adaptation à ce contexte.

## *VI.2 Approche multi-modèles*

Pour répondre à ces problématiques, nous avons proposé une approche multi-modèles. Dans un premier temps, notre approche présente un modèle de programmation générique, intégrant des mécanismes de communications synchrones, asynchrones, ainsi que des patrons de conception permettant des communications plus particulières. Ce modèle de programmation se base sur les annotations Java, afin de séparer clairement le code métier des directives de répartition. Il permet également d'optimiser l'utilisation asynchrone des objets distants, en modifiant automatiquement le code d'appel pour découpler le fil d'exécution de l'appelant et de l'appelé.

Dans un deuxième temps, cette approche multi-modèles présente une combinaison de modèles de communication, pour adresser les problèmes induits par la mobilité. Ainsi l'utilisation possible de plusieurs modèles de communication permet l'interopérabilité avec des composants utilisant différents intergiciels. De plus, l'utilisation conjointe et simultanée d'une combinaison de modèles permet de s'adapter au contexte, afin d'opter pour le modèle le plus pertinent face à la situation.

Dans un troisième temps, notre approche présente des mécanismes d'adaptation dynamique. Ces mécanismes sont basés d'une part sur le chargement et la modification à la volée des classes annotées, permettant un chargement dynamique de nouvelles classes inconnues lors du développement.
Ils sont d'autre part basés sur des politiques d'adaptation, modifiables à la volée lors de l'exécution, et spécifiant le ou les modèles de communication à utiliser (et dans quel ordre) en fonction des observations de contexte fournies par des observateurs.

## *VI.3 Implémentations et résultats*

Pour valider et implémenter notre approche multi-modèles, nous avons dans un premier temps développé un greffon multi-modèles, voué à être utilisé

avec un proxy Internet embarqué dans un train. Le problème adressé ici est réel : les communications avec le sol ne peuvent être continues.
Appliquant une partie de notre approche à la résolution du problème, nous avons développé un greffon, capable de modifier le comportement de toute application Java. Ainsi les communications `Socket` peuvent être changées en transmissions de messages JMS lorsque le contexte le justifie.
Des évaluations et comparaisons avec le système standard ont été effectuées, démontrant ainsi la pertinence d'un tel système multi-modèles pour résoudre des problèmes liés à la mobilité.

Nous avons ensuite conçu et développé une version statique de l'infrastructure logicielle multi-modèles, i.e. les points clés de notre approche ont été implémentés, mais les mécanismes d'adaptation dynamique comme la modification des politiques ou le chargement et la modification des classes à la volée n'ont pas été implantés.
L'infrastructure résultante de ce développement fournit un moyen rapide et efficace de permettre la répartition d'une application : une fois spécifiées par les annotations (ou explicitement codées par les méthodes fournies par l'infrastructure), les communications sont intégrées à l'application lors d'une étape de compilation intermédiaire, la rendant ainsi autonome pour pouvoir s'exécuter directement, et bénéficier ainsi de l'approche multi-modèles pour établir ses communications.

L'infrastructure logicielle multi-modèles "complète" a enfin été implémentée, fournissant les mécanismes dynamiques autorisant la reconfiguration à la volée de l'infrastructure et de l'application.

## *VI.4 Perspectives*

Si notre infrastructure est aujourd'hui utilisable et son intérêt pertinent, certaines perspectives d'améliorations et d'ajouts restent néanmoins présentes. Ainsi, comme nous l'avons exposé précédemment, nous y avons intégré deux modèles de communication, RMI et JMS. Des modèles additionnels pourraient d'une part permettre une plus grande interopérabilité, et d'autre part une adaptation au contexte plus fine et plus performante. C'est pourquoi l'implantation de modèles comme JavaSpaces, ou les Web Services est envisagée pour compléter notre infrastructure.

Le modèle de programmation fournit d'ores et déjà des patrons de conception "espace partagé", "file d'attente de messages" et "sujet d'intérêt". L'en-

richissement de ce modèle par l'ajout de patrons comme les "événements" ou bien l'amélioration des patrons existants (notamment l'"espace partagé" qui demeure relativement basique) rendra ce modèle, et donc l'infrastructure, plus utile et performant pour les développeurs.

L'évaluation du greffon multi-modèles a chiffré le coût de notre approche en matière de performances, mais a également justifié son utilisation en mobilité. En complément de cette analyse, une évaluation comparative de notre infrastructure logicielle multi-modèles appuiera cette justification, et quantifiera le bénéfice apporté par notre approche au développement d'applications réparties.

Enfin, l'utilisation de notre infrastructure pour implémenter des cas concrets de développement et d'utilisation d'applications réparties illustrera notre approche, et démontrera par l'exemple l'intérêt de nos travaux.

# Références


[AWE+08] Artenum, EBM WebSourcing, Edifixio, INRIA, INT, IRIT, Obeo, and Open Wide. SCOrWare project. http ://www.scorware.org, 2008.

[Bak01] Sean Baker. A2a, b2b - now we need m2m (middleware to middleware) technology. Proceedings of the Third IEEE International Symposium on Distributed-Objects and Applications (DOA'01), 2001.

[Bal04] Roland Balter. JORAM : Java Open Reliable Asynchronous Messaging. http ://joram.objectweb.org, 2004.

[BB02] Victor Budau and Guy Bernard. Changement dynamique de modèle de communication dans une plate-forme logicielle pour composants. Journée Systèmes à composants adaptables et extensibles, 2002.

[BCA+01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and K. Saikoski. The design and implementation of openorb v2. IEEE Distributed Systems Online, vol 2 no 6, Special Issue on Reflective Middleware, 2001.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An open component model and its support in java. Lecture Notes in Computer Science, pages 7–22, 2004.

[BCTT05] Antoine Beugnard, Olivier Caron, Jean-Philippe Thibault, and Bruno Traverson. Assemblage de composants par contrats : Le

modèle de composants accord. L'objet, vol. 11, num 4, pages 11–36, 2005.

[BEK+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. http ://www.w3c.org/TR/SOAP/, 2000.

[BG06] Aurélien Bocquet and Christophe Gransart. Les modèles de communication - état de l'art. Technical Report INRETS/RR-06-730 FR, Projet REVE, 2006.

[BG07a] Aurélien Bocquet and Christophe Gransart. Comunication everywhere : Internet proxy adapted to satellite communication defects for railway transport application. Intelligent Transport Systems and Services, 2007.

[BG07b] Aurélien Bocquet and Christophe Gransart. Multi-model architecture for its software design improvements. ITS Telecommunications, 2007.

[BG08a] Aurélien Bocquet and Christophe Gransart. Approche et solution multi-modèles aux déconnexions satellitaires pour l'accès internet à bord des trains. Journées des Applications Spatiales, Toulouse Space'Show, 2008.

[BG08b] Aurélien Bocquet and Christophe Gransart. Benefits of multi-model architecture-based development for railway applications. World Congress on Railway Research, 2008.

[BGG06] Aurélien Bocquet, Christophe Gransart, and Jean-Marc Geib. Application de l'architecture multi-modèles pour la gestion des déconnexions satellitaires à bord des trains. UbiMob - Ubiquité et Mobilité, actes, 2006.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm : un outil de manipulation de code pour la réalisation de systèmes adaptables. Journée Composants, Grenoble, Proceedings, 2002.

[BLFFN96] Timothy John Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext transfer protocol, http/1.0. RFC 1945, 1996.

[Boc06] Aurélien Bocquet. Architecture multi-modèles pour l'accès à des services en mobilité. Journée des doctorants de l'INRETS, 2006.

[Boc07] Aurélien Bocquet. De l'approche à l'architecture multi-modèles. Journée des doctorants de l'INRETS, 2007.

[Boc08] Aurélien Bocquet. Infrastructure logicielle multi-modèles pour l'accès à des services en mobilité. EuroDoc'Info 2008, 2008.

[CE05] FP6 Commission Européenne. Projet InteGRail : INTElligent inteGration of RAILway systems. http ://www.integrail.info, 2005.

[Cha07] David Chappell. Introducing sca. A White Paper by David Chappell, 2007.

[CKM+03] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next step in web services. ACM Communications in Service oriented computing, vol. 46, issue 10, pages 29–34, 2003.

[CT94] Chih-Ping Chu and Chi-Jen Tzeng. On the development paradigm of distributed applications. International Conference on Parallel and Distributed Systems, pages 736–741, 1994.

[DHDTS98] Bruno Dumant, Francois Horn, Frédéric Dang Tran, and Jean-Bernard Stéfani. Jonathan : an open distributed processing environment in java. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing : Middleware'98, 1998.

[DM08] Franck Desbuquois and Julien Merlin. Tests et évaluation d'un greffon de communication multi-modèles. Rapport de projet Master I, 2008.

[Ern07] T Ernst. Network Mobility Support Goals and Requirements, RFC 4886. http ://www.faqs.org/rfcs/rfc4886.html, 2007.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. JavaSpaces Principles, Patterns, and Practice. Pearson Education, 1999.

[GARG02] Christophe Gransart, Sébastien Ambellouis, Jean Rioult, and Uster Guillaume. Objets mobiles communicants et transports. Colloque sur la mobilité, LORIA, Nancy, 2002.

[GB08] Christophe Gransart and Aurélien Bocquet. Modèles de modes de communication. Technical Report REVE, document D1.2.2, Projet REVE, 2008.

[GBS03] Paul Grace, Gordon S. Blair, and Sam Samuel. Remmoc : A reflective middleware to support mobile client interoperability. International Symposium of Distributed Objects and Applications (DOA), Proceedings, 2888 :1170–1187, 2003.

[GGM99] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. CORBA, des concepts à la pratique. Dunod, 1999.

[Gho00] Debalina Ghosh. Mobile ip. Crossroads, vol 7, issue 2, 2000.

[GL06] Jean-Luc Grimault and Pierre Lévis. Connexion à internet depuis un réseau embarqué : analyse d'après les rôles métiers. UBIMOB, 3e journées francophones Mobilité et Ubiquité, 2006.

[Hal85] Robert H. Jr. Halstead. Multilisp : a language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 7, issue 4, pages 501–538, 1985.

[HHD98] Richard Hayton, Andrew Herbert, and Douglas Donaldson. Flexinet - a flexible, component oriented middleware system. 8th ACM SIGOPS European Workshop, Proceedings, 1998.

[JCLC05] INRIA Jacquard, INSA CITI, INRETS LEOST, and CNAM CEDRIC. REVE, safe Reuse of Embedded components in heterogeneous enviRonmEnts, projet ANR ARA-SSIA. http ://www.ara-reve.org, 2005.

[KDRB91] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, CCristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Proceedings of the 11th European Conference on Object-Oriented Programming, 1997.

[Kra06] Sacha Krakowiak. Introduction à l'intergiciel. Ecole d'été sur les Intergiciels et sur la Construction d'Applications Réparties, Autrans, 2006.

[Kra08] Sacha Krakowiak. Middleware Architecture with Patterns and Frameworks. Ebook under Creative Commons license, 2008.

[KRL+00] Fabio Kon, Manuel Roman, Ping Liu, Jina Mao, Tomonori Yamane, Luis Claudio Magalhaes, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. ACM Middleware'2000, 2000.

[Lam86] Leslie Lamport. On interprocess communication. Distributed Computing, 1986.

[LSR+] INRETS LEOST, SNCF, France Telecom R&D, GET, CISCO, and THALES. Train-IPSat, projet PREDIT. http ://www.inrets.fr/ur/leost/projets/trainipsat.htm.

[Mic02] Sun Microsystems. JMS, Java Message Service. http ://java.sun.com/products/jms, 2002.

[Mic03] Sun Microsystems. RMI, java Remote Method Invocation. http ://java.sun.com/products/jdk/rmi, 2003.

[Mic04] Sun Microsystems. Package java.lang.instrument : provides services that allow Java programming language agents to instrument programs running on the JVM. http ://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html, 2004.

[Mic06a] Sun Microsystems. Entreprise JavaBeans specification 3.0, JSR 220. http ://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[Mic06b] Sun Microsystems. JSR 224 : JavaTM API for XML-Based Web Services (JAX-WS) 2.0. http ://jcp.org/en/jsr/detail ?id=224, 2006.

[MLB04] Juliette Marais, Sebastien Lefebvre, and Marion Berbineau. Satellite propagation path model along a railway track for gnss applications. Vehicular Technology Conference, VTC2004-Fall, pages 4066–4070, 2004.

[MWN02] Scott McLean, Kim Williams, and James Naftel. Microsoft .Net Remoting. Microsoft Press, 2002.

[NR68] Peter Naur and Brian Randell. Software Engineering : Report of a conference sponsored by the NATO Science Committee. Scientific Affairs Division, NATO, Brussels 39 Belgium, 1968.

[OO87] David J. Otway and Ed Oskiewicz. Rex : a remote execution protocol for object-oriented distributed applications. Proceedings of the 7th International Conference on Distributed Computing Systems, 1987.

[Pau02] Laurent Pautet. Intergiciels schizophrènes : une solution fortement interopérable fondée sur les composants adaptables. Systèmes à composants adaptables et extensibles, 2002.

[Paw05] Renaud Pawlak. Spoon : annotation-driven program transformation — the aop case. ACM Proceedings of the 1st workshop on Aspect oriented middleware development, 2005.

[Paw06] Renaud Pawlak. Spoon : Compile-time annotation processing for middleware. IEEE Distributed Systems Online, vol. 7, no. 11, 2006.

[PDF+02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. JAC : An Aspect-Based Distributed Dynamic Framework, 2002.

[Pos80a] Jon Postel. Transmission control protocol. RFC 761, 1980.

[Pos80b] Jon Postel. User datagram protocol. RFC 768, 1980.

[QKP99] Thomas Quinot, Fabrice Kordon, and Laurent Pautet. CIAO. http ://adabroker.eu.org/ciao, 1999.

[QKP01] Thomas Quinot, Fabrice Kordon, and Laurent Pautet. From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models. International Symposium of Distributed Objects and Applications (DOA), Proceedings, 2001.

[Qui03] Thomas Quinot. Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables. Thèse de doctorat, Université Pierre et Marie Curie, Paris, 2003.

[RAG04] Jean Rioult, Sebastien Ambellouis, and Christophe Gransart. Système de détection de l'arrêt souhaité pour un usager des transports en commun (ICAU). Brevet déposé, 2004.

[Riv96] Fred Rivard. Smalltalk : a reflective language. Reflection'96, 1996.

[Sar05] Project-Team Sardes. System architecture for reflective distributed computing environments. Technical report, INRIA Rhônes-Alpes, 2005.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. Component Software - Beyond Object-Oriented Programming - Second Edition. Addison-Wesley / ACM Press, 2002.

[SOS+06] Cliff Schmidt, Daryl Olander, Davanum Srinivas, Eddie O'Neil, Ken Tam, Kyle Marvin, and Rich Feit. Apache's Beehive Project. http ://beehive.apache.org, 2006.

[TBO05] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. Qinna, a component-based qos architecture. Lecture Notes in Computer Sciences : Component-Based Software Engineering, pages 107–122, 2005.

[Tus08] Apache Tuscany. Tuscany SCA Java 1.3. http ://tuscany.apache.org, 2008.

[VHPK04] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Polyorb : a schizophrenic middleware to build versatile reliable distributed applications. Lecture Notes in Computer Science, 3063 :106–119, 2004.

[YB04] Jia Yu and Rajkumar Buyya. A novel architecture for realizing grid workflow using tuple spaces. Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, pages 119–128, 2004.

# ANNEXE A

## Différentes annotations fournies par l'infrastructure logicielle multi-modèles

L'infrastructure logicielle multi-modèles fournit un certain nombre d'annotations afin de permettre simplement au développeur de rendre des objets répartis, d'en utiliser, ou de spécifier un certain nombre de paramètres concernant la répartition ou l'invocation.

Les annotations fournies ici représentent une liste non exhaustive des annotations possibles dans l'infrastructure, car de nouvelles annotations sont ajoutées au modèle de programmation au fur et à mesure des besoins ressentis lors de l'implémentation et de l'utilisation de l'infrastructure, et les annotations listées ici représentent simplement un instantané des possibilités offertes par le modèle de programmation à ce jour.

La notion d'événements a été citée dans notre approche, mais comme nous l'avons déjà précisé, aucune implémentation du principe n'a encore été intégrée à l'infrastructure logicielle multi-modèles. De même, les annotations permettant la définition et l'utilisation d'événements n'ont pas encore été intégrées. Ceci étant, leur utilisation sera semblable aux sujets d'intérêts, seuls les paramètres seront spécifiques au principe d'événements.

## *A.1 Annotations pour les objets et méthodes "serveur"*

**DistributedClass**
Cette annotation destinée à qualifier une classe la désigne comme potentiellement exportable via l'infrastructure : les instances d'objets de cette classe pourront être accessibles à distance.
Pour qu'une classe soit effectivement exportable, il faut qu'une méthode au moins de celle-ci soit définie comme distribuée (DistributedMethod).

Le paramètre possible de cette annotation est :
- type (default : "Sync") : spécifie le type de comportement emprunté par cette classe. Peut être l'un des types "Sync", "Async", "OneWay".

**DistributedObject**
Cette annotation destinée à qualifier un attribut intialisé d'une classe DistributedClass l'exporte pour le rendre accessible via l'infrastructure.

Les paramètres possibles de cette annotation sont :
- name (default : "") : spécifie le nom symbolique sous lequel l'objet sera référencé auprès du service de nommage. Si aucun nom n'est spécifié, l'objet n'est pas référencé.
- registered (default : false) : spécifie si l'objet doit être référencé auprès du service de nommage. Le paramètre name doit être également précisé pour que l'objet soit effectivement référencé.

**DistributedMethod**
Cette annotation destinée à qualifier une méthode d'une classe DistributedClass la rend accessible à distance via l'infrastructure. Elle est ignorée si le type de DistributedClass est autre que "Sync", "Async" ou "OneWay".

Le paramètre possible de cette annotation est :
- type (default : "") : spécifie le type d'invocation que le client distant effectuera sur la méthode. Ce paramètre n'a aucune incidence sur le comportement de l'objet serveur, il ne sert qu'à spécifier le type d'invocation au client. Peut être l'un des types "" (utilise le type de DistributedClass), "Sync", "Async", "OneWay".

**DistributedAttribute**
Cette annotation destinée à qualifier un attribut d'une classe DistributedClass le rend accessible à distance par le biais de méthode *get* et *set* acces-

sibles via l'infrastructure. Elle est ignorée si le type de DistributedClass est autre que "Sync", "Async" ou "OneWay".

Le paramètre possible de cette annotation est :
- access (default : "RW") : spécifie le type d'accès à l'attribut exporté via l'infrastructure. Peut être l'un des types "R" (lecture), "W" (écriture), "RW" (contrôle total).

## *A.2 Annotations pour les objets et méthodes appelants ("client")*

**ResolveObject**
Cette annotation destinée à qualifier un attribut d'une classe quelconque instancie un objet proxy client permettant l'appel aux méthodes exportées de l'objet DistributedObject désigné.

Les paramètres possibles de cette annotation sont :
- name (default : "*") : spécifie le nom désignant l'objet réparti référencé auprès du service de nommage. Peut être un nom exact ou comportant des méta-caractères '*' ou ' ?'. Si une méthode FilteringMethod prenant en charge le type d'objet a été désignée, elle sera appelée pour lui soumettre tous les candidats référencés.
- location (default : "") : spécifie l'emplacement du service de nommage sur lequel effectuer la recherche d'objet distant. La chaine vide désigne le serveur local.
- type (default : "") : spécifie le type de modèle de programmation à utiliser pour invoquer les méthodes de l'objet distant. Peut être l'un des types "" (utiliser le type défini pour chaque méthode dans la classe DistributedClass), "Sync" (appel synchrone), "Async" (appel asynchrone), "OneWay" (appel unidirectionnel). Lorsque le type est redéfini dans cette annotation, toutes les méthodes de l'objet sont appelées selon le même modèle.

**FilteringMethod**
Annotation prévue mais non encore implémentée. Dans le cas de l'instanciation d'un objet proxy représentant un objet distant, son nom symbolique ou sa référence peuvent être spécifiés, mais la sélection peut également être faite dynamiquement. Le développeur désigne alors une méthode de filtrage `boolean method(<E> remoteObject)` par cette annotation, et tous les "candidats" possibles à l'instanciation sont soumis à cette méthode afin qu'elle

sélectionne l'objet voulu. La méthode doit prendre en paramètre le type d'objet distant voulu. Comme tous les mécanismes inhérents à ce système de filtrage ne sont pas encore complètement définis, cette annotation reste pour l'instant un projet.

**ForceSync**
Cette annotation destinée à qualifier une méthode d'une classe quelconque précise que tous les appels à des méthodes d'objets distants dans cette méthode devront être synchrones. Cette annotation est prioritaire sur les paramètres de ResolveObject, DistributedMethod et DistributedClass.

**ForceOneway**
Cette annotation destinée à qualifier une méthode d'une classe quelconque précise que tous les appels à des méthodes d'objets distants dans cette méthode devront être unidirectionnels. Cette annotation est prioritaire sur les paramètres de ResolveObject, DistributedMethod et DistributedClass.

**ForceAsync**
Cette annotation destinée à qualifier une méthode d'une classe quelconque précise que tous les appels à des méthodes d'objets distants dans cette méthode devront être asynchrones. Cette annotation est prioritaire sur les paramètres de ResolveObject, DistributedMethod et DistributedClass.

## *A.3 Annotations pour les patrons de conception*

### *A.3.1 MessageQueuePattern*

Les files de messages peuvent être utilisées de deux façons : soit par l'utilisation d'un objet MessageQueue fourni par l'infrastructure, soit en utilisant ces annotations. La file d'attente créée par annotation ne peut que recevoir des messages, et non en émettre.

**MessageQueuePattern**
Cette annotation destinée à qualifier une classe la désigne comme potentiellement utilisable comme file de messages par l'infrastructure : les instances d'objets de cette classe (si annotés DistributedObject) pourront être accessibles pour recevoir des messages.
Pour qu'une classe soit effectivement exportable, il faut qu'une méthode QueueReceiveMethod soit définie.

**QueueConnectMethod**
Cette annotation destinée à qualifier une méthode de type `boolean method(String id, String pwd)` d'une classe MessageQueuePattern permet de déclarer une méthode d'autorisation d'utilisation de cette file.

**QueueReceiveMethod**
Cette annotation destinée à qualifier une méthode de type `void method(Object o)` d'une classe MessageQueuePattern permet de déclarer la méthode qui sera appelée à la réception d'un message dans cette file.

### *A.3.2 TopicPattern*

Les sujets d'intérêt peuvent être utilisés de deux façons : soit par l'utilisation d'un objet Topic fourni par l'infrastructure, soit en utilisant ces annotations. Le sujet d'intérêt créé par annotation ne peut que recevoir des messages, et non en émettre.

L'utilisation de sujets d'intérêt ou de files de messages par annotations ne présente pour l'instant aucune différence notable, excepté leur accessibilité par un intergiciel présentant nativement ces concepts : e.g. un sujet d'intérêt annoté peut être utilisé directement par une application utilisant JMS et se connectant au sujet d'intérêt JMS correspondant.

**TopicConnectMethod**
Cette annotation destinée à qualifier une méthode de type `boolean method(String id, String pwd)` d'une classe TopicPattern permet de déclarer une méthode d'autorisation d'utilisation de ce sujet.

**TopicReceiveMethod**
Cette annotation destinée à qualifier une méthode de type `void method(Object o)` d'une classe TopicPattern permet de déclarer la méthode qui sera appelée à la réception d'un message dans ce sujet.