N° d'ordre : 2883

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Pierre VIGNÉRAS**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones.

Soutenue le : 08 novembre 2004

Après avis des rapporteurs :

Françoise Baude	Maître de Conférences
Doug Lea	Professor, USA
Michel Riveill	Professeur

Devant la commission d'examen composée de :

Françoise Baude	Maître de Conférences	Rapporteur
Serge Chaumette	Professeur	Directeur de thèse
Olivier Coulaud	Directeur de Recherche	Président
Mohamed Mosbah	Maître de Conférences	Examinateur
Alexis Moussine-Pouchkine	Consultant Sun	Examinateur
Michel Riveill	Professeur	Rapporteur

– 2004 –

Remerciements

Il me semble important de remercier l'ensemble des personnes qui m'ont soutenu durant cette thèse.

Le monde de la recherche

Dans le cadre professionnel tout d'abord, l'ensemble de cette aventure n'aurait pu être vécue sans la proposition de sujet de thèse de Serge Chaumette à une période charnière de mon existence : la sortie du service militaire ! De nombreuses questions ont été soulevées à l'époque, et Serge m'a fait part de son expérience et de son avis. Sur ce point, Asier Ugarte, alors doctorant dans l'équipe a lui aussi été de bon conseil.

Sur la recherche proprement dite, je veux souligner les nombreux échanges que j'ai eu au début avec Serge et Asier. Le travail en équipe était réel : des réunions hebdomadaires nous permettaient d'avancer en tenant compte de l'avis d'un groupe. L'arrivée de Pascal Grange a encore renforcé l'équipe. Ses critiques "cinglantes" facilitaient la remise en question. Mais c'est le départ d'Asier qui m' en a directement rapproché. Pascal est devenu peu à peu mon interlocuteur privilégié, mon consultant, puis finalement, mon ami. Dans les périodes de doutes dont tout thésard est la proie, Il a toujours su me convaincre de l'intérêt de mes travaux tout en remettant en cause les solutions logicielles, algorithmiques ou formelles que je pouvais trouver. Vraiment, je souhaite à tous les thésards d'avoir un "Pascal Grange" comme voisin de bureau ! Si les discussions sont souvent longues – voire interminable, au grand damne de nos conjointes respectives, – elles sont toujours fructueuses : Pascal ne parle jamais pour ne rien dire !

Je tiens aussi à signaler que j'ai toujours eu le soutien de Marie Beurton-Aimar, maître de conférences à Bordeaux II. Elle s'est toujours intéressée à mes travaux et d'une manière plus générale à mon avenir. J'en suis très touché, j'ai particulièrement apprécié nos échanges. Je regrette que notre collaboration n'ait pu aboutir.

Parmi les contacts qui m'auront marqué durant ces années, Georges Eyrolles, alias *Georgy* – maître de conférences à l'ENSEIRB – tient une place à part. J'ai beaucoup de considération pour ses qualités d'enseignant et j'ai énormément appris à son contact dans le domaine du génie logiciel. La découverte de la *programmation extrême*, et son adaptation à un cadre pédagogique ont été, pour moi, une révolution. En outre, j'ai pu juger de la pertinence de ces principes en appliquant un certain nombre de ces règles au développement de la plate-forme Mandala.

Je remercie aussi énormément mes rapporteurs d'avoir eu le courage de lire un si volumineux ouvrage. Le retour trés rapide et les nombreuses critiques – toujours constructives – de Françoise Baude m'ont aidé à soigner mon document rapidement. Son rapport trés détaillé m'a touché profondément : j'ai en effet pris conscience

du conséquent travail qu'elle a du réaliser. Je n'oublie pas non plus Doug Lea – éminent professeur américain – qui m'a ouvert les yeux sur l'évolution marquante de la communauté des agents mobiles vers les supports d'exécution asynchrones. En outre, j'estime beaucoup les échanges par courrier éléctronique – distance oblige – que je continue d'avoir avec eux.

Je termine cette partie en remerciant l'ensemble de mon jury de soutenance de thèse d'avoir écouté mon monologue de 45 minutes et d'avoir posé les questions qui m'ont permis de prendre du recul sur mes travaux et d'envisager l'avenir avec enthousiasme.

Cadre privé

Dans le cadre privé, l'ensemble de mes relecteurs peut être félicité : si le manuscrit a été apprécié, c'est aussi grâce aux multiples corrections qu'ils m'ont demander d'apporter sans me froisser ! Et pourtant, il s'agit d'une véritable corvée : la lecture d'un document de thèse dont on ne connaît pas le domaine est entâchée de termes barbares, de sigles cabalistiques et de figures ésotériques. Il faut du courage pour se contenter de ne corriger que les fautes de français ! A ce propos, Philippe Lacoue – mon éternel dévoué – a joué un rôle non négligeable : il a été jusqu'à la correction des annexes, parties trés formelles de ce document !

D'une manière plus générale, ma famille et mes amis ont été à l'écoute de mes soucis lorsque j'en avais besoin. Si Pascal – pour être du domaine – comprenait le mieux mes problèmes, les autres n'ont pas manqué à mes appels : ma mère et ma marraine en particulier ont toujours cru en moi ; ma sœur et mon frére se sont souvent préoccupés de mes travaux ; Angélique et Laurent se sont régulièrement intéressés à ma situation. Je n'oublierai ni les personnes qui sont venues assistées à ma soutenance, ni celles qui, pour une raison ou une autre, n'ont pas pu venir. Sachez que ce jour là fût pour moi un moment chargé d'émotions. Je vous en suis trés reconnaissant.

De l'importance du conjoint

J'ai volontairement éclipsé dans cette partie le socle, la base de mon équilibre : ma fiancée, Séverine, alias *Sigma*. Sa grande discrétion cache une volonté farouche de me voir réussir. Ces dernières années ont été particulièrement difficile, pour de multiples raisons. Elle a su me protéger d'un certain nombre de problèmes pour m'aider à tenir le cap, à me concentrer sur mon travail. Elle a aussi accepté les contraintes temporelles liées au métier de l'enseignement et de la recherche.

En outre, durant cette thèse, elle m'a donné deux adorables petits garçons : Louis et Thomas. Aprés l'épisode douloureux du premier accouchement, elle a su

prendre les reines de la famille ce qui a facilité mon implication dans mes travaux. L'arrivée de Thomas aurait pu compliquer les choses, mais, en dépassant les difficultés matérielles et organisationnelles, elle est devenue une mère attentive, tout en restant la merveilleuse femme que j'ai toujours connu. Par exemple, elle sait me faire prendre conscience des moments où je dois décrocher de mes passions, pour être présent au sein de ma famille, sans me brusquer. C'est ainsi que par leur seule présence, mes enfants me font relativiser l'importance de mes travaux.

Tout cela contribue à mon équilibre : un épanouissement simultané dans la vie privée et dans la vie professionnelle !

L'esprit de groupe

A la vue de ces remerciements, on peut se rendre compte de l'importance de l'entourage professionnel, amical et familial dans la conduite d'une thèse jusqu'à son terme. Je n'oublierai pas toutes ces années : ni les moments douloureux ni les bonheurs intenses. Surtout, je garderai le souvenir de tout ceux qui m'ont soutenu.

Cette thèse est donc l'aboutissement d'un travail collectif. Si elle doit être considérée comme une réussite, alors, c'est *notre* réussite. Chacun peut alors être un peu fier de lui !

Merci à tous !

Résumé

I Précision du domaine

Le **développement d'une application distribuée** est toujours une opération complexe à réaliser demandant de fortes compétences dans plusieurs domaines : génie logiciel, protocoles réseaux, parallélisme, sécurité, etc. Alors que les premières applications étaient essentiellement des architectures *clients/serveurs* dont les principales problématiques étaient liées à la sécurité et à l'extensibilité, la tendance actuelle est aux applications fortement distribuées dans lesquelles un fragment logiciel peut agir simultanément comme client et comme serveur. Par exemple un composant peut être le client d'une base de données pour servir une requête HTTP. Si les problèmes précédents sont toujours au cœur des préoccupations des programmeurs, la facilité de développement est un facteur qui est loin d'être négligeable tant le coût d'un logiciel dépend du temps de développement et de débogage. C'est pourquoi des abstractions ont été introduites au fil des années dans le but de faciliter la programmation.

Parmi les nombreux exemples, nous pouvons distinguer au niveau du langage, la notion d'objets qui a nettement amélioré le génie logiciel des applications et au niveau des bibliothèques, l'abandon progressif de la manipulation directe des sockets UNIX au profit d'abstractions de type *appel de procédure à distance* comme les SUN-RPC, Java-RMI ou la couche de communication de CORBA. Même dans le domaine du calcul haute performance où le standard MPI - basé sur les deux appels de très bas niveaux `send()` et `receive()` - est encore très utilisé, des outils de haut niveaux existent : High Performance Fortran par exemple est encore employé par les non-informaticiens en raison de sa syntaxe simplifiée et de son adéquation à la résolution de problèmes à parallélisme de données. Plus récemment, l'environnement PM2, à base d'appels de procédures à distance et de processus légers a apporté un niveau d'abstraction supplémentaire facilitant la mise en œuvre de mécanismes d'équilibrage de charge dans les applications irrégulières massivement parallèles. A l'autre extrême, dans les systèmes distribués hétérogènes tels que les architectures multi-couches utilisées dans les gros serveurs *web* par exemple, les composants logiciels sont les abstractions qui garantissent un faible coût de développement et de

maintenance. C'est dans ce cadre que l'on retrouve l'environnement `.NET` ou les *Enterprises Java Beans* (EJBs).

Le domaine de nos travaux est celui des systèmes distribués, et en particulier des abstractions logicielles au niveau du langage ou du modèle de programmation qui facilitent le développement d'applications distribuées.

II Cadre de travail

Nos travaux s'inscrivaient au départ dans le cadre du développement d'une plate-forme distribuée, multi-utilisateurs et multi-tâches appellé JEM – Experimentation environMent for Java. Celle-ci permet à l'utilisateur et au programmeur de manipuler de manière homogène n'importe quel type de ressource matérielle ou logicielle connectée à un réseau. Les ressources sont organisées par le support d'exécution de manière arborescente de telle sorte qu'elles apparaissent à l'utilisateur final comme un système de fichiers de type UNIX. Cependant, la notion d'entité d'exécution, inspirée par les processus dans les systèmes UNIX, n'existe pas dans JEM. L'objectif initial de nos travaux étaient donc de rechercher une telle entité d'exécution, en prenant en compte la notion de *mobilité*. JEM étant une plate-forme distribuée, l'entité d'exécution devait pouvoir migrer d'une machine virtuelle Java à une autre.

Nous nous sommes donc intéressés à cette abstraction particulière du domaine des systèmes distribués – le code mobile – et en avons effectué une modélisation d'un système d'agents mobiles. Cette modélisation a fait apparaître un nouveau concept : *le conteneur actif*. Cette nouvelle abstraction a également été modélisée et nous en avons réalisé une implémentation en Java : JACOb. L'étude de ce concept et de son implémentation a soulevé le problème de la gestion de la concurrence dans les applications distribuées. Par conséquent, nous avons étudié les moyens d'exprimer la concurrence, et nous proposerons une nouvelle alternative : *la référence asynchrone* et son implémentation appelée RAMI. L'ensemble de ces travaux ont abouti à une solution logicielle fonctionnelle et documentée pour le développement d'applications concurrentes et/ou distribuées : Mandala.

III Mobilité

Le code mobile a un intérêt particulier pour le développeur car il facilite l'écriture des fragments de code d'une application distribuée qui gére des problèmes récurrents (connexion lente et/ou temporaire, équilibrage de charge, tolérance aux pannes, etc.).

III.1 Etat de l'art

Cette thèse dresse donc **un état de l'art du domaine de la mobilité** en en précisant la terminologie et les paradigmes employés : le *code à la demande*, l'*évaluation à distance* et les *agents mobiles*. L'état de l'art ira jusqu'à prendre en compte les *processus légers mobiles* et les *vecteurs de codes mobiles* que sont les *cartes à microprocesseurs*. Nous constaterons que c'est un domaine de recherche en pleine effervescence puisqu'un certain nombre de problèmes - notamment ceux liés à la sécurité - restent encore ouverts.

III.2 Modélisation

Dans le cadre de cette étude, nous avons établi une modélisation du paradigme "agents mobiles" en π-calcul. Nous verrons que l'aspect "mobile" peut être modélisé par la notion abstraite de *vue* : un agent n'est *visible* que du serveur qui le contient. C'est ce même serveur qui "exporte" la vue de l'agent aux autres entités du système global. Cette notion de *contenance* est d'une importance fondamentale. Elle nous amènera vers la définition d'**une nouvelle abstraction : le conteneur actif**.

III.3 Notre contribution : le *conteneur actif*

Cette abstraction est assez simple : elle est définie comme un conteneur capable d'invoquer les méthodes d'un objet qu'il contient pour le compte d'un tiers. Ce concept permet de considérer la quasi-totalité des applications à objets distribués comme des cas particuliers d'un système à conteneurs actifs. En l'occurrence, nous verrons que les systèmes d'agents mobiles peuvent s'exprimer assez simplement à l'aide de cette nouvelle abstraction. De même, la gestion des objets distribués est facilitée par l'utilisation de conteneurs actifs. D'un certain point de vue, les conteneurs apportent de manière intrinsèque une *séparation des aspects* (aspect distant et aspect métier) d'un objet distribué. Surtout, cette abstraction apporte du *dynamisme* dans l'utilisation des objets distribués – caractéristique fondamentale des systèmes à migration de code.

Nous présenterons notre implémentation de ce concept en Java : JACOb (Java Active Container of Objects). Nous verrons que cette plate-forme offre un support pour la communication et le placement dynamique d'objets et nous la comparerons avec des outils similaires, tels que Java-RMI, Corba, ProActive, JavaParty, les EJBs ou encore JavaSpace.

III.4 Perspectives

Finalement, un certain nombre de perspectives seront envisagées comme, par exemple, l'utilisation des conteneurs comme modèle mémoire ou l'ajout de services (nommage, sécurité, monitoring, ...) pour faciliter le développement d'applications distribuées.

IV Gestion de la concurrence

Lors de l'utilisation de notre implémentation du concept, nous avons été confrontés à un problème de performance lié à la gestion de la concurrence. Nous avons donc étudié les difficultés liés à l'expression de la concurrence dans un programme – et plus particulièrement dans un langage orienté objets.

IV.1 Etat de l'art

Dans les applications distribuées, la *concurrence* est une caractéristique fondamentale qui peut être à l'origine de problèmes variés (deadlocks, incohérence de l'état des objets) ou d'optimisation non-négligeable (parallélisme ou recouvrement calcul/communication). Nous verrons que les threads posent de nombreux problèmes dans le développement d'applications orientées objets (choix de la politique concurrente, portabilité) ; et que leur utilisation est généralement inadaptée pour exprimer la concurrence dans les langages orientés objets (différenciation entre l'objet représentatif et le flot d'exécution). Enfin, en Java, une certaine gymnastique syntaxique doit être utilisée, laissant généralement les débutants assez perplexes. Par ailleurs, la notion de thread n'est pas absolument nécessaire pour exprimer la concurrence et nous verrons plusieurs alternatives : modèle évenementiel, objets actifs, modèle acteurs, objets séparés (modèle concurrent de B. Meyer dans Eiffel).

La manipulation des objets contenus dans les conteneurs actifs – appelés *stored objects* – nous a amenés à définir un type particulier de référence. Ce type permet de gérer la concurrence des appels de méthodes des *stored objects*. Nous verrons que ce type est une spécialisation d'un type plus général que nous proposons comme **une nouvelle abstraction : la référence asynchrone**.

IV.2 Notre contribution : la *référence asynchrone*

Cette abstraction est l'extension "naturelle" de la référence standard sur des objets en Java que nous appellerons *référence synchrone*. Elle permet l'invocation asynchrone de toutes les méthodes de l'objet qu'elle référence. Ainsi, en considérant

les appels de méthodes comme généralement asynchrones, cette abstraction facilite le développement des applications orientées objets qui exploitent la concurrence d'exécution. Par ailleurs, elle s'étend relativement bien au cas distant, facilitant l'écriture des applications distribuées, notamment celles qui utilisent le modèle de conteneur actif. Nous présenterons notre implémentation de ce concept en Java : RAMI (Reflective Asynchronous Method Invocation). Nous verrons que son utilisation permet d'intégrer la concurrence de manière relativement naturelle et transparente en masquant la *politique de gestion de la concurrence* à l'utilisateur final (une thread par appel, utilisation d'un *threadpool*, ordonnancement FIFO, etc.). Tous les objets manipulés au travers de références asynchrones ne devront cependant pas avoir la même politique de gestion de la concurrence afin d'éviter des problèmes d'innocuités.

Nous montrerons aussi que **la transparence des appels est problématique dans le cas asynchrone**. Nous proposerons une **solution intermédiaire : la semi-transparence** qui offre aux développeurs un cadre habituel de programmation (appel de méthode) tout en évitant le problème principal de la transparence totale : la méconnaissance de la sémantique asynchrone de ses appels.

Nous présenterons enfin un comparatif avec des travaux similaires tels que ProActive, RRMI ou encore le mécanisme d'appel de méthode asynchrone de Corba.

IV.3 Perspectives

Nous proposerons plusieurs pistes à explorer pour le développement d'applications concurrentes (distribuées ou non) en utilisant le modèle des références asynchrones. Par exemple, le développement d'un compilateur optimiseur de code utilisant les références asynchrones ou la modification d'une machine virtuelle pour qu'elle supporte directement le concept de référence asynchrone.

V Organisation de la thèse

Cette thèse s'articule donc autour des *conteneurs actifs* et des *références asynchrones*. Ces deux abstractions forment un tout cohérent pour le développement d'applications distribuées orientées objets. Ces travaux ont naturellement débouché sur une implémentation appelée Mandala qui permet aux développeurs Java de bénéficier des deux abstractions que nous avons définies. Cette implémentation écrite en Java est opérationnelle, documentée et disponible en licence libre LGPL sur SourceForge : `http://mandala.sf.net/`.

Nous présenterons aussi brièvement les quelques applications qui ont été développées à l'aide de cette plate-forme et les implications qu'elles ont eu sur son développement ou sur les concepts sous-jacents.

Abstract

I Domain overview

The **development of a distributed application** is a complex operation requiring strong knowledges in several areas: software engineering, network protocols, parallelism, security, etc. Whereas the first distributed applications were primarily based on *client/server* architectures whose principal problems were related to security and scalability, the current trend is to design highly distributed applications in which a software component can act simultaneously as a client and as a server. For example a component can be the client of a database to serve a HTTP request. Even though the preceeding problems are still the main concerns of the programmers, the ease of development is far from being negligible since the cost of a software strongly depends on the time needed for its development and its debugging. This is why abstractions to make programming easier have been introduced through years.

Among the many examples of such improvements, we can distinguish:

at the language level, the concept of object which has clearly improved the software engineering of applications;

at the libraries level, the progressive abandonment of the low level UNIX socket API in favor of the *remote procedure call* abstraction, such as the Sun-RPC, Java-RMI or the communication layer of CORBA.

Even in the field of high performance computing where the standard MPI – based on the two very low level calls `send()` and `receive()` – is still widely used, high level tools exist: High Performance Fortran for example is (still) practiced by scientists thanks to its simple syntax and its good fit to data parallel program. More recently, the PM2 environment which is based on remote procedure calls and on light weight processes, brought an additional level of abstraction making easier the development irregular highly parallel applications. This evolution is also observed in the domain of heterogeneous distributed systems such as multi-tier architectures used in large web servers. Software components are the abstractions which guarantee a low maintenance cost and a low development cost. This is where we find the Example frameworks are `.NET` and *Enterprises Java Beans* (EJBs).

The scope of our work is that of distributed systems, particularly the software abtractions at the language level or the programming models which make the development of distributed applications easier.

II Surround of this work

Our work originaly took part in the development of a distributed, multi-user and multi-task platform, called JEM for Experimentation environMent for Java. This platform allows the user and the programmer to handle any type of hardware or software resource connected to a network in a homogeneous way. Resources are organized as a tree structure by the runtime system then appearing to the end-user like a UNIX file system. However, the concept of execution unit, inspired by the UNIX process abstraction, does not exist in JEM. The initial and first goal of our work was thus to design such an execution unit, taking into account the concept of *mobility*. JEM being a distributed platform, the execution unit was to be able to migrate from a node (a Java virtual machine) to another one.

We were thus interested in the particular abstraction of the field of the distributed systems that is mobile code and we carried out a modeling of a mobile agents system. This modeling revealed a new concept: *the active container*. We also modelled this new abstraction and we achieved an implementation in Java that we called JACOb. The study of this concept and of its implementation naturally raised the problem of the management of concurrency in distributed applications. Consequently, we considered the means of expressing concurrency, and we propose a new alternative, *the asynchronous reference*. Its implementation is called RAMI. This work resulted in a fully functional and well documented software solution for the development of concurrent and/or distributed applications: Mandala.

III Mobility

Mobile code is particularly interesting for the developers because it makes it easier to write the code fragments of a distributed application which handle recurrent problems such as slow and/or temporary connection, load balancing, fault-tolerance, etc.

III.1 State of the art

In this thesis we draw up **a state of the art of the field of mobility** precisely defining the terminology and the paradigms (*code on demand*, *remote evaluation* and *mobile agents*). The state of the art also takes into account *mobile light weight*

processes and the *mobile code vectors* that are *smartcards*. It is a very active field of research because a certain number of problem, in particular those related to security, remain open.

III.2 Our first contribution: a formal model of mobile agents

We modelled the mobile agents paradigm using π-calculus. We show that the mobile aspect can be modelled by the abstract concept of *sight*: an agent is *visible* only by the server that contains it. It is this same server that exports the sight of the agent to the other entities of the global system. This concept of *containment* is fundamental. This led us to define **a new abstraction: the active container**.

III.3 Our second contribution: *the active containers*

The active containers are a rather simple abstraction: they are defined as containers able to invoke the methods of an object that they contain for the account of a third party. This concept makes it possible to regard almost every distributed object oriented application as a particular case of an active container based system. For instance, we show that mobile agents systems can be expressed easily using this new abstraction. The management of distributed objects is also made easier by the use of active containers. From a certain point of view, containers intrinsically bring a *separation of aspects* (remote and business aspect) to distributed objects. Especially, this abstraction brings *dynamism* in the use of distributed objects – a fundamental characteristic of mobile code systems.

We present our implementation of this concept in Java that we called JACOb for Java Active Container of Objects. This platform provides a support for communication and dynamic placement of objects. We compare it to similar tools, such as Java-RMI, Corba, ProActive, JavaParty, EJBs or JavaSpace.

III.4 Future work

We are currently considering a certain number of prospects like the use of containers as a memory model or the addition of services (naming, security, monitoring, etc.) to ease the development of distributed applications.

IV Management of concurrency

When using our implementation of the active container concept, we were confronted to a problem of efficiency in the management of concurrency. We thus studied

the problems related to the expression of concurrency in a program and more particularly in an object oriented program.

IV.1 State of the art

Concurrency is a fundamental characteristic of distributed applications which can lead to various problems (deadlocks, objects inconsistency) or to significant optimizations (parallelism or computation/communication overlapping). Threads are one of the most widely used paradigms to handle concurrency. We will see however that this paradigm leads to many problems in the development of object oriented applications (concurrent policy choice, portability) and that it is generally unsuited to express concurrency in object oriented languages (differentiation between the object representative and the flow of execution). Furthermore, in Java, the syntax that must be used, generally leaves beginners rather perplex. In addition, the concept of thread is not absolutely necessary to express concurrency and we will see several alternatives: event-based models, active objects, actors, separate objects (concurrent model of B. Meyer in Eiffel).

The management of the objects stored into active containers – called *stored objects* – leads us to define a particular reference type. This type allows the handling of the concurrency in stored objects methods invocations. We will see that this reference type is a specialization of a more generic type that we propose as **a new abstraction: the asynchronous reference**.

IV.2 Our third contribution: *the asynchronous references*

Asynchronous references are the natural extension of standard Java references. It allows the asynchronous invocation of all the methods of the object that it references. Thus, by regarding the calls of methods as generally asynchronous, this abstraction eases the development of objects oriented applications that take advantage of the concurrency of execution. In addition, it extends relatively well to the remote case, making it easier to write distributed applications, in particular those which use the active container model. We present our implementation of this concept in Java that we called RAMI for Reflective Asynchronous Method Invocation. We show that it makes it possible to deal with concurrency in a relatively natural and transparent way by masking the *concurrent policy* to the end-user (one thread per call, threadpool, FIFO scheduling, etc). However, all the objects handled through asynchronous references should not have the same concurrent policy in order to prevent safety problems.

We also show that the **transparency of the calls is a problem in the**

asynchronous case. We therefore propose an **intermediate solution: semi-transparency**. This offers the developers the usual paradigm of method invocation still avoiding the major issue of total transparency (hidden asynchronous semantic of the calls).

We eventually compare our work to similar projects such as ProActive, RRMI or the mechanism of asynchronous method invocation of Corba.

IV.3 Future work

Several directions seem to be worth exploring for the development of concurrent applications (distributed or not) using the model of asynchronous references. For example, the development of an optimizing compiler using asynchronous references or a modification of a virtual machine so that it directly supports asynchronous reference.

V Organization of the thesis

This thesis is organized around our contributions, i.e. *active containers* and *asynchronous references*. These two abstractions are coherent as a whole for the development of distributed object oriented applications. This work led to an implementation called Mandala which makes it possible for Java developers to effectively use the two abstractions that we defined. This Java implementation is operational, documented and available under the LGPL open-source licence on SourceForge: `http://mandala.sf.net/`.

We also present the applications developed using this platform and the implications they had on its development or on the underlying concepts.

Table des matières

Liste des tableaux

Table des figures

Liste des programmes

Première partie

Introduction générale

Chapitre 1

Introduction

1.1 Domaine visé

Les applications parallèles ont longtemps été exécutées sur des machines sophistiquées, très performantes (IBM [54] RS/6000 [44], Cray [104] T3E [45], Intel [55] Paragon [67]) mais aussi très onéreuses. Ces machines ont été conçues pour le calcul intensif qui reste un domaine marginal comparé au nombre d'applications fonctionnant sur l'ensemble du parc de machines actuelles.

Les performances actuelles des micro-ordinateurs, liées aux progrès réalisés par les nouveaux matériels de communication, facilitent l'accès à la puissance de calcul. Le projet Beowulf [27] par exemple utilise des *clusters* de stations pour le calcul intensif. Surtout, la 23 ème édition du Top500 des supercalculateurs [136] (juin 2004), présente un tournant historique : les systèmes labélisés *clusters* sont majoritaires (ils sont 291) [135]. En France, le projet Grid5000 [10] vise à construire une *grille* de calculs en reliant par un réseau très haut débit des *clusters* de plusieurs régions.

Pour ce qui est du logiciel, un certain nombre d'applications nécessitent une forte puissance de calcul et génèrent du parallélisme sans pour autant entrer dans le domaine du calcul intensif. Par exemple, les serveurs HTTP doivent répondre à un grand nombre de requêtes et, pour chaque requête, ils peuvent être amenés à interroger un serveur de base de données, à tracer un graphique, à générer une page HTML, etc. Le parallélisme sous-jacent peut être pris en compte par l'utilisation d'une machine dédiée à chaque tâche (serveur de base de données, serveur graphiques, serveur de pages). Nous parlerons d'applications distribuées pour différencier ce type d'architecture des architectures dédiés aux calculs intensifs : les problématiques ne sont généralement pas les mêmes. Dans les premières, la disponibilité et la sécurité du système sont les critères parmi les plus importants. Dans les supercalculateurs, la performance est l'argument de vente (nombre d'opérations par secondes et débit réseaux).

Les applications distribuées sont très répandues aujourd'hui au sein des entreprises qui ont une vitrine sur Internet. Elles sont généralement développées à l'aide de composants objets d'où la notion d'applications à objets distribuées.

Dans le même temps, le langage Java [20, 89] n'est plus seulement considéré comme un langage de programmation d'*applets* sur Internet mais est utilisé pour le développement d'applications généralistes, particulièrement dans le domaine des objets distribués [194]. En effet, du fait de sa conception autour d'une machine virtuelle (*Java Virtual Machine*) permettant l'exécution d'un code intermédiaire (le *bytecode* Java), il est très approprié pour le développement et le déploiement d'applications dans un environnement hétérogène. On évoque souvent le problème de la lenteur de son exécution liée à l'interprétation du bytecode. Quelques solutions ont été trouvées, parmi lesquelles : les compilateurs à la volée (*Just In Time Compilers*) [183] qui traduisent le *bytecode* en code natif au moment de l'exécution, les compilateurs natifs [39, 106] qui génèrent un exécutable à partir du *bytecode* (GCJ, *GNU Java Compiler* [199], est le plus populaire dans le monde UNIX) et la technologie *HotSpot* [193] de Sun qui effectue une compilation à la volée conditionnelle[1].

Enfin, Java est devenu un standard de fait que l'on retrouve dans tous les domaines de l'informatique : JavaCard [143] pour les cartes à microprocesseurs, Java 2 Micro Edition (J2ME) [196] pour les systèmes embarqués, Java 2 Standard Edition (J2SE) [198] pour les stations de travails, Java 2 Enterprise Edition (J2EE) [195] pour les serveurs d'entreprises. Des rencontres internationales ont même lieu régulièrement au titre du *Java Grande Forum* [14] pour soumettre des changements au langage ou à la machine virtuelle Java ou pour standardiser des bibliothèques afin de faciliter le développement d'applications parallèles en Java.

Par ailleurs, même si dans le domaine du calcul intensif les environnements MPI [93] et, dans une moindre mesure maintenant, PVM [84], restent les deux standards de communication, dans le domaine des applications à objets distribués, ce sont CORBA [186] et RMI [191] (et DCOM [138] dans le monde Microsoft) qui l'emportent.

Notre équipe, forte d'une expérience dans le domaine du calcul parallèle, s'est orientée vers l'amélioration du développement d'applications distribuées (génie logiciel, facilité de déploiement, généralisation aux réseaux hétérogènes non dédiés, etc.). A ce titre, Java offre un certain nombre d'avantages par rapport à ses rivaux[2] : il intègre des constructions qui permettent d'exprimer la concurrence (`java.lang.Thread`, `synchronized`, `wait()` et `notify()`), il offre un format qui facilite son déploiement sur des environnements hétérogènes et il propose un mécanisme

[1] *i.e.* seule une partie du code est compilée. Le choix étant effectué par une étude dynamique.

[2] Ceci est un peu moins vrai aujourd'hui, les langages C# [139] et Ruby [13] par exemple sont des alternatives intéressantes.

d'appel de méthode à distance relativement simple : RMI (*Remote Method Invocation*). En outre, il bénéficie d'un support communautaire important, de plusieurs implémentations de machine virtuelles alternatives (Microsoft [141][3], IBM [101], Kaffe [7], Jikes RVM [102], Classpath [9], etc...) et d'une riche API [197].

1.2 Sujet de thèse

Le sujet initial de cette thèse était d'étudier le domaine du code mobile à la recherche d'une entité migrable pour la plate-forme appelée JEM [209, 46, 208], développée au sein de notre équipe. JEM est une machine virtuelle multi-utilisateurs et distribuée qui organise les ressources matérielles et logicielles de manière arborescente de telle sorte qu'elles apparaissent à l'utilisateur final comme un système de fichiers de type UNIX. Par contre, la notion d'entité d'exécution, inspirée par les processus dans les systèmes UNIX, n'existe pas. L'objectif initial de mes travaux était donc de rechercher une telle entité d'exécution, en prenant en compte la notion de *mobilité*. JEM étant une plate-forme distribuée, l'entité d'exécution devait pouvoir migrer d'une machine à une autre. En outre, une modélisation des systèmes d'agents mobiles devait être réalisée afin de prouver un certain nombre de propriétés (atteignabilité d'un agent destinataire d'un message, retour au serveur d'origine, utilisation des agents mobiles dans un cadre de tolérance aux pannes ou d'équilibrage de charge dynamique, etc.). Un système d'agents mobiles capable d'interopérer avec la plate-forme JEM, ou mieux, d'en être la couche de base devait ensuite être implémenté.

Une étude de la migration de code a donc été réalisée et fait l'objet de la partie II de ce document. Dans cette étude, nous nous sommes focalisés plus particulièrement sur le paradigme *agents mobiles* que nous avons modélisé.

Cette modélisation, réalisée en π-calcul, et présentée en annexe C, a fait apparaître une nouvelle structure de données active que nous avons appelée *conteneur actif*.

Elle semble être une entité de bas niveau au-dessus de laquelle un système d'agents mobiles peut être créé. Mieux cette entité propose de fait une implémentation *dynamique* du mécanisme d'appels de méthodes à distance : les objets n'ont pas à être prévus pour être accessibles de manière distante. Par ailleurs, ce concept favorise l'exploitation du parallélisme, en rendant les appels de méthodes implicitement *asynchrones*. La partie III est entièrement dédiée à la présentation de ce nouveau concept et à notre implémentation de celui-ci : JACOb.

En exploitant le concept des conteneurs actifs de manière distante, la gestion de

[3]Qui n'est plus délivrée en raison de l'accord passé entre Microsoft et Sun en Avril 2004 [140].

la concurrence s'est avérée nécessaire. Après avoir étudié les modèles d'expressions, nous avons cherché une solution qui s'intègre bien au concept de nos conteneurs actifs. Nous proposons à cet effet une nouvelle abstraction, la *référence asynchrone*, qui permet d'accéder à tout objet de manière asynchrone. La partie IV présente cette nouvelle abstraction ainsi que notre implémentation : RAMI.

L'ensemble de ces deux concepts, les *conteneurs actifs* et les *références asynchrones* forme un tout cohérent pour le développement d'applications concurrentes ou distribuées (et parallèles). Cet ensemble est présenté sous la forme d'un projet nommé Mandala [212], documenté, disponible en licence libre LGPL sur le site Sourceforge : `http://mandala.sf.net/`. Des mesures de performances ainsi que des applications seront présentées au chapitre V qui est consacré à Mandala.

Enfin, nous verrons en conclusion les améliorations possibles de la plate-forme et les perspectives envisagées.

Chapitre 2

Contributions

La lecture de cette thèse ne donnerait une vision globale de nos travaux qu'à la fin du document, lors de la présentation de Mandala en §V p199. Afin d'éviter de perdre le lecteur dans les méandres de nos travaux (état de l'art, détails d'implémentations, mauvais choix, etc.) il nous semble important de donner tout de suite un aperçu de nos contributions. Dans un premier temps, les contributions seront replacées dans un ordre chronologique qui n'est pas tout à fait respecté par ce document. Ensuite, quelques exemples de programmes écrits avec Mandala donneront une idée de ses possibilités. Enfin, un comparatif de notre plate-forme avec différents travaux similaires que nous avons étudiés fera l'objet d'une section à part entière.

2.1 Chronologie

L'organisation de ce document ne reflète pas tout à fait la réelle chronologie de nos travaux pour des raisons de cohérence dans le plan (l'aspect formel se trouve dans les annexes). Aussi, nous dressons ci-dessous la liste de nos contributions dans l'ordre chronologique avec pour chacune d'elle, la partie de ce document qui s'y réfère.

1. État de l'art du domaine de la mobilité (§II p27)
2. État de l'art des formalismes utilisés (§A p273)
3. Modélisation d'un système d'agents mobiles (§C.3 p297)
4. Nouvelle abstraction : les conteneurs actifs (§III p75)
5. Modélisation du concept des conteneurs actifs (§D p313)
6. Implémentation en Java de ce concept (JACOb) (§10 p95)
7. État de l'art du domaine de la concurrence (§12 p123)
8. Nouvelle abstraction : les références asynchrones (§14 p147)

9. Implémentation en Java de ce concept (RAMI) (§15 p157)
10. Distribution de l'ensemble en un tout cohérent sous forme de plate-forme logicielle open-source (Mandala [212]) facilitant le développement d'applications orientées objets concurrentes et/ou distribuées (§V p199)

2.2 Exemples

Cette section donne quelques exemples de programmes écrits en Java qui illustrent les possibilités de Mandala. Les caractéristiques de Mandala– et la terminologie qui leurs sont associées – sont présentées au fur et à mesure.

2.2.1 HelloWorld

Pour ne pas déroger à la tradition, nous présentons le classique *hello world*[1]. Dans un premier temps, nous écrivons la classe `HelloWorld.java`.

```
1  package helloWorld;
2  public class HelloWorld implements java.io.Serializable {
3      public String toString() { return "Hello World !"; }
4  }
```

Le programme principal se réduit à une méthode `main()` qui a été décomposé pour mettre en évidence les différentes étapes :

```
1  public static void main(String[] args) {
2      helloWorld.HelloWorld helloWorld = new helloWorld.HelloWorld();
3      String s =  helloWorld.toString();
4      System.out.println(s);
5  }
```

Le version asynchrone (et sans intérêt) de ce programme s'écrit :

```
1  public static void main(String[] args) throws Throwable{
2      // Gets an asynchronous proxy on a new helloWorld.HelloWorld object
3      jaya.helloWorld.HelloWorld helloWorldProxy =
4                                          new jaya.helloWorld.HelloWorld();
5      // Asynchronous call
6      mandala.rami.FutureClient future = helloWorldProxy.rami_toString();
7      doSomethingElse();
8      String s = (String) future.waitForResult();
9      System.out.println(s);
10 }
```

[1] Répertoire `Examples/src/helloWorld/` de la distribution de Mandala [214].

A ce stade, il faut faire plusieurs remarques afin de comprendre ce code :

- les classes dont le nom est préfixé par `jaya.` sont des proxy asynchrones semi-transparents (§16.2 p186) générés par le compilateur `jayac` (§16.2.3 p189) ; ils encapsulent une référence asynchrone (§14 p147) pour en faciliter l'utilisation ;
- les méthodes dont le nom est préfixé par `rami_`[2] sont asynchrones, elles renvoient un objet *future* de type `FutureClient` (§15.1.1 p158) qui offre un mécanisme de récupération *active* (§14.4.1 p152) du résultat.

Enfin, la version distante du programme précédent s'écrit :

```
1  import mandala.rami.Framework; // RAMI provides asynchronism
2  import mandala.jacob.ActiveMap; // JACOb provides active containers
3  import mandala.jacob.remote.RemoteActiveMap; // and also remote ones!
4  ...
5  public static void main(String[] args) throws Throwable {
6  ------------------------------ Framework Initialisation ------------------------------
7      // gets a proxy on a remote active map using JNDI
8      ActiveMap activeMap = (RemoteActiveMap)
9                      new javax.naming.InitialContext().lookup(args[0]);
10
11     // New asynchronous references will be created by the SORFactory
12     // which provides remote asynchronous reference
13     Framework.setFactory(new mandala.jacob.SORFactory(activeMap));
14 ------------------------------ End of Initialisation ------------------------------
15     // Common part with the local case
16     jaya.helloWorld.HelloWorld helloWorldProxy =
17                                          new jaya.helloWorld.HelloWorld();
18     mandala.rami.FutureClient future = helloWorldProxy.rami_toString();
19     doSomethingElse();
20     String s = (String) future.waitForResult();
21     System.out.println(s);
22 }
```

Là encore, plusieurs remarques sont nécessaires:

- l'interface `ActiveMap` (PROG. 15.5 p161) définit un conteneur actif (§8.2 p76), elle hérite de l'interface `java.util.Map` ;
- l'interface `RemoteActiveMap` définit un conteneur actif distant, elle hérite de `ActiveMap` et de `mandala.jacob.Remote` qui définit le mécanisme de gestion des pannes réseaux (§10.2.3 p100) ;
- Mandala utilise *Java Naming and Directory Interface*[144] (JNDI) pour la récupération de références distantes ce qui lui permet d'être indépendant du protocole utilisé et en particulier du *registry* de RMI ;

[2]RAMI est l'acronyme de *Reflective Asynchronous Method Invocation.*

- le type de références asynchrones créé par défaut est paramétrable par l'intermédiaire de la classe `Framework` ;
- la classe `SORFactory` créée des références asynchrones sur des objets contenus[3] dans le conteneur actif qui lui est associé ;
- le code "métier" est exactement le même dans la version locale et dans la version distante.

Ces exemples très simples permettent de définir le domaine d'application principal de Mandala :

- **la concurrence (§IV p123) :** les méthodes des objets peuvent être invoquées de manière asynchrones ;
- **la distribution (§9 p79) :** les objets peuvent être accédés à distance ;

En outre, ces exemples illustrent les caractéristiques principales de Mandala :

- **le dynamisme (§10.5.1 p112) :** les classes n'ont pas besoin d'être spécifiques à Mandala pour être utilisées de manière asynchrone et/ou distante ;
- **l'unification :** en utilisant le modèle de programmation à base de références asynchrones (§9.11.1 p91), le développement d'applications distribuées n'est pas différent du développement d'applications locales.

2.2.2 Écritures asynchrones

Considérons le code suivant :

```
1  import java.io.*;
2  ...
3      final File file = ...
4      final FileWriter writer = new FileWriter(file);
5      try{
6          while(true) {
7              final String s = getStringToWrite();
8              if (s == null) break;
9              writer.write(s);
10         }
11
12         doSomethingElse();
13
14         // Really write
15         writer.flush();
16     }finally {
17         if (writer != null) writer.close();
18     }
```

[3] Appelés *stored objects* et référencés de manière asynchrone par les instances de la classe `StoredObjectReference` (§10.4 p111).

Ce genre de code est très souvent rencontré dans les applications Java qui manipulent des entrées/sorties. On s'intéresse dans cet exemple au temps nécessaire pour sortir de la boucle `while()`) et atteindre la méthode `doSomethingElse()`. Diminuer ce temps peut être réalisé en utilisant une sortie bufferisée à l'aide d'une encapsulation de type `java.io.BufferedWriter`. Mais cela ne suffit pas. Il faut en effet adapter la taille du *buffer* en fonction de celle des chaînes à écrire. Si celles-ci sont importantes, le buffer n'amène rien puisqu'il peut être rempli à chaque écriture. Il apporte même un surcoût. Une étude du code est donc nécessaire ce qui n'est pas toujours possible si la méthode `getStringToWrite()` dépend de données dynamiques. Une alternative asynchrone utilisant Mandala offre une solution élégante :

```
1  import java.io.*;
2  import mandala.rami.impl.AsynchronousPolicyFactory;
3  ...
4      final File file = ...
5  ____________________________ Prepare the instanciation ____________________________
6      // Must use a FIFO asynchronous policy
7      final AsynchronousPolicyFactory apf = new FifoAPFactory();
8      // This factory will create asynchronous references with a FIFO
9      // asynchronous policy
10     final Framework.Factory factory = new ARFactory(apf);
11 ______________________________ End of Preparation ______________________________
12     final jaya.java.io.Writer writer =
13                                 new jaya.java.io.FileWriter(file, factory);
14     try{
15         while(true) {
16             final String s = getStringToWrite();
17             if (s == null) break;
18                 writer.rami_write(s); // Asynchronous call
19         }
20
21         doSomethingElse();
22
23         // Really write
24         writer.flush(); // Synchronous call
25     }finally {
26         if (writer != null) writer.close(); // Synchronous call
27     }
```

Notons qu'il y a peu de différence dans le code métier (dont l'instanciation ne fait pas directement partie) entre les deux versions (mise en évidence par une police **grasse**). Pour la compréhension de ce code, nous devons à nouveau faire plusieurs remarques :

- le typage des proxy asynchrones semi-transparents est symétrique de celui

de leur classe synchrone (ici, la classe `java.io.FileWriter` implémente `java.io.Writer` et il en est de même de leur proxy respectifs générés par le compilateur `jayac` (§16.2.3 p189)) ;
- le type de références asynchrones créé peut être paramétré à l'instanciation de son proxy asynchrone semi-transparent associé ;
- la politique asynchrone (§15.2.3.1 p167) utilisée est paramétrable[4] (ici, une gestion FIFO est nécessaire pour assurer un ordre correct des écritures) ;
- un proxy asynchrone semi-transparent permet aussi d'effectuer des appels synchrones ; mais dans ce cas, la référence asynchrone associée doit être utilisée – même de manière cachée – pour garantir une sémantique correcte (appel distant et/ou application d'une politique asynchrone FIFO par exemple).

Ces deux derniers points sont d'une importance fondamentale pour la cohérence de notre version asynchrone. En effet, c'est parce que dans cet exemple, la politique asynchrone employée est de type FIFO que l'exécution des méthodes `flush()` et `close()` n'ont lieu qu'une fois toutes les ecritures asynchrones réalisées. Cependant, Mandala propose d'autres politiques asynchrones (une *thread* par appel ou utilisation d'un *thread pool* par exemple) et il est assez simple d'en écrire de nouvelles. Il faut par contre les utiliser à bon escient.

Nous avons mesuré les performances de la solution asynchrone[5]. Pour éviter de polluer les mesures avec la génération de la chaîne de caractères, la méthode `getStringToWrite()` retourne toujours la même chaîne s dont la taille en octets $|s|$ est fixée au début de manière aléatoire. Cette chaîne est retournée n fois. On mesure le temps δ nécessaire pour sortir de la boucle `while()` et le temps total Δ nécessaire pour passer l'instruction `close()`. On mesure l'*efficacité* E par le rapport $E = \frac{\Delta}{\delta}$. Une efficacité élevée, signifie que le flot d'exécution est sorti de la boucle `while()` bien avant que toutes les écritures sur disque soient achevées. Le débit D moyen de ces écritures est donné par : $D = \frac{|s|*n}{\Delta}$

Nous souhaitons comparer les versions synchrones et asynchrones, aussi, nous définissons le *speedup* S par :

$$S = \frac{E_{asynchrone}}{E_{synchrone}}$$

et l'overhead O par :

$$\Delta_{asynchrone} = \Delta_{synchrone} + \Delta_{synchrone}.O \Leftrightarrow O = \frac{\Delta_{asynchrone}}{\Delta_{synchrone}} - 1$$

[4]Mais il dépend du type de références asynchrones utilisé. En particulier les références asynchrones créées par un `SORFactory` ont des politiques asynchrones fixées par le conteneur via lequel elles sont accédées. Il est toutefois possible de paramétrer indépendemment la politique du conteneur (une fois pour toute) et celle des *stored objects* par l'intermédiaire d'un chaînage de référence (§15.3 p172).

[5]Fichier `Examples/src/io/AIO.java` de la distribution de Mandala [214].

Le test a été effectué 100 fois sur un bi-Pentium III (katmai) à 450 MHz disposant de 512 Mo de RAM. L'écriture se fait sur le répertoire temporaire standard `/tmp/`, donc sur le disque local qui est de type SCSI II. La machine virtuelle est celle livrée avec le JRE v1.4.2_03 de Sun. Les données sont les suivantes :

$$10 \text{ Ko} \leq |s| \leq 20 \text{ Ko} , \qquad 1000 \leq n \leq 2000$$

La ligne de commande est : `java io.AIO 100` (pour 100 exécutions du test).

Les résultats sont présentés dans le tableau TAB. 2.1 p14. Seule la moyenne est prise en considération pour lisser l'effet du *Just In Time compiler* et du ramasse-miettes[6]. On le voit, pour un overhead de l'ordre de 3%, la version asynchrone atteint la sortie de la boucle 850 fois plus rapidement que dans le cas synchrone. L'efficacité E dans le cas synchrone est constante, et égale à 1. Même en utilisant un `BufferedWriter`, associé à un tampon de taille conséquente (1 Mo), l'efficacité reste égale à 1. C'est un peu étonnant est n'avons pas encore trouvé la raison de ce comportement. Par conséquent, nous avons :

$$S = \frac{E_{asynchrone}}{E_{synchrone}} = \frac{E_{asynchrone}}{1} = \frac{\Delta_{asynchrone}}{\delta_{asynchrone}}$$

Si cet exemple met en lumière les performances de notre version asynchrone, il faut surtout remarquer la simplicité du code proposé. Nous laissons au lecteur le soin d'écrire une version asynchrone à base de *threads*, et de la comparer à celle-ci.

Notons enfin qu'en changeant simplement le type de références asynchrones l'écriture peut se faire à distance (en utilisant par exemple un `SORFactory`).

2.2.3 Programmation graphique

Mandala permet de développer selon un modèle événementiel (§12.3.1 p126). Ce style de programmation reste incontournable dans les applications graphiques en général, et dans celles écrites en Java à base de composants Swing en particulier (§12.3.3.1 p132). Considérons le fragment de code suivant :

```
1  import java.util.Collection;
2  import java.util.Iterator;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  ...
6      final JPanel panel = new JPanel();
7  ...
8      final Collection c = ...;
```

[6] L'appel `System.gc()` est tout de même effectué avant chaque test.

Nombre d'appels (n)	1487
Taille de la chaîne ($\|s\|$)	15.5 Ko
Taille totale ($\|s\| * n$)	23 Mo
Débit (D)[a]	850 Ko/s[b]
Speedup (S)	854
Overhead (O)	2.82 %

[a] Le débit est mentionné ici pour information dans le cas synchrone.

[b] Les faibles performances s'expliquent par le grand nombre d'appels réalisés et le passage de l'encodage standard Unicode à celui de la plate-forme sous-jacente.

TAB. 2.1 – Performances des entrées/sorties asynchrones.

```
9     final Iterator i = c.iterator();
10    while(i.hasNext()) {
11        final Object o = i.next();
12        final JLabel label = new JLabel();
13        String s = o.toString();
14        label.setText(s);
15        panel.add(label);
16    }
```

L'idée est d'afficher les éléments d'une collection graphiquement. Ici, c'est la méthode `toString()` qui est affichée, mais cela pourrait être une autre méthode (commune à tous les objets évidemment). Le problème survient lorsque les objets de la collection sont des références vers des objets distants. Dans ce cas, l'appel à distance de la méthode `toString()`[7] peut prendre un certain temps. Pour pallier à ce problème, une approche simple et efficace consiste à utiliser une approche événementielle :

- chaque label est initialisé avec une chaîne de caractère qui indique à l'utilisateur que le résultat n'est pas encore disponible (par exemple "Waiting...") ;
- la méthode `toString()` est invoquée à distance de manière asynchrone ;
- lorsque le résultat est disponible, le texte du label est mis à jour.

Cette solution s'écrit très simplement avec Mandala comme le montre le code suivant :

```
1  import java.util.Collection;
2  import java.util.Iterator;
```

[7] Notons qu'avec RMI, il n'est pas possible d'invoquer la méthode "métier" directement à moins d'utiliser un mécanisme compliqué faisant intervenir les proxy dynamiques du JDK v1.3.

```
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import mandala.rami.Callback;
6  ...
7      final JPanel panel = new JPanel();
8  ...
9      final Collection c = ...;
10     final Iterator i = c.iterator();
11     while(i.hasNext()) {
12         // Each object in the collection is an asynchronous proxy
13         final jaya.java.lang.Object o = (jaya.java.lang.Object) i.next();
14         final JLabel label = new JLabel("Waiting...");
15         o.rami_toString(new Callback() {
16             public void done(InvocationInfo info, MethodResult result) {
17                 final String s;
18                 try{
19                     s = (String) result.getReturnedResult();
20                 }catch(Throwable t) {
21                     s = t.getMessage();
22                 }
23
24                 label.setText(s);
25                 label.revalidate();
26                 label.repaint();
27             }
28           });
29         panel.add(label);
30     }
31 ...
```

Le mécanisme de *callback*s utilisé dans Mandala (§15.1.2 p160) permet donc des écritures relativement naturelles. Ce code est à comparer à celui qui utilise les threads Java ou même les objets *future*s (§15.1.1 p158).

2.2.4 Programmation distribuée

Cet exemple présente la machinerie interne de Mandala qui est masquée à l'utilisateur et montre les possibilités offertes dans le domaine de la programmation distribuée.

Nous avons vu dans le premier exemple que l'aspect *dynamique* permet à tout objet d'être accessible à distance en étant inséré dans un conteneur actif distant. Le lecteur attentif pourrait penser que cela n'est vrai que pour les objets sérialisables. Comment en effet insérer un objet non sérialisable dans un conteneur distant. La solution proposée par Mandala est assez élégante : un service générique (§10.6 p114)

d'instanciation à distance est fourni par le biais de la classe `mandala.jacob.Instantiator` qui contient – entre autres – la méthode suivante :

```
// Returns the key mapped to the freshly instanciated stored object
public Object instanciate(final java.lang.reflect.Constructor constructor,
                          final Object[] args);
```

Chaque instance de cette classe est nécessairement associée à un conteneur actif (distant ou non). Cette classe étant sérialisable, il est donc possible d'associer une de ses instances `instantiator` à un conteneur actif distant `activeMap`, d'insérer `instantiator` dans `activeMap` et d'en récupérer une référence asynchrone `sor` sur le *stored object* `instantiator` :

```
import mandala.jacob.ActiveMap;
import mandala.jacob.StoredObjectReference;
import mandala.jacob.Instantiator;
import mandala.jacob.remote.RemoteActiveMap;
...
    // gets a proxy on a remote active map using JNDI ActiveMap
    activeMap = (RemoteActiveMap)
          new javax.naming.InitialContext().lookup(args[0]);
    Instantiator instantiator = new Instantiator(activeMap);
    // Try to get a unique key
    Object key = String.valueOf(System.identityHashCode(instantiator));

    // Inserts instantiator into the remote activeMap
    activeMap.put(key, instantiator);
    // After its insertion, instantiator has a *local* reference on activeMap
    // which is still remote for us

    // Gets an asynchronous reference on it
    StoredObjectReference sor =
                StoredObjectReference.getInstance(activeMap, key);
```

Ensuite, on peut utiliser cette référence asynchrone pour instancier à distance un objet, l'insérer dans un conteneur actif et en récupérer la clé. Par commodité, nous utiliserons un proxy asynchrone semi-transparent :

```
import java.lang.reflect.Constructor;
import java.io.File;
import java.io.FileWriter;
...
    // Gets a semi-transparent asynchronous proxy on it
    jaya.mandala.jacob.Instantiator instantiatorProxy =
                    jaya.mandala.jacob.Instantiator.getInstance(sor);
```

```
8
9       // We will create the file /tmp/foo.txt remotely
10      File file = new File("/tmp/foo.txt");
11      Constructor constructor =
12          FileWriter.class.getConstructor(new Class[] {File.class});
13      // Synchronous remote call
14      key = instantiatorProxy.instanciate(constructor, new Object[] {file});
```

Remarquons que la méthode `instanciate()` est invoquée de manière synchrone. Lorsque plusieurs objets doivent être instanciés à distance, la version asynchrone `rami_instanciate()` peut apporter un gain considérable surtout si elle est associée à un *callback* qui utilise l'objet dès son insertion (utilisation au plus tôt).

Finalement, on récupère une référence asynchrone sur ce *stored object* nouvellement créé ce qui permettra de l'utiliser à distance et de manière asynchrone :

```
1   import mandala.rami.FutureClient;
2   import mandala.jacob.remote.TransportException;
3   ...
4       // Gets an asynchronous reference on it
5       sor = StoredObjectReference.getInstance(activeMap, key);
6
7       // Gets a semi-transparent asynchronous proxy on it
8       jaya.java.io.FileWriter writer =
9                       jaya.java.io.FileWriter.getInstance(sor);
10
11      // Use it!
12      FutureClient future = writer.rami_write("Asynchronous and remote write!");
13      try{
14          // May throw IOException and TransportException (a RunnableException)
15          writer.write("Synchronous, remote (maybe concurrent) write!");
16          // May throw Throwable
17          future.waitUntilResultAvailable();
18      }catch{IOException ioe) {
19          // As usual!
20      }catch(TransportException te) {
21          // A remote exception occured!
22      }catch(Throwable t) {
23          // Anything else (Error, Runtime, ...)
24      }
```

Si le deuxième appel à la méthode `write()` est synchrone il peut potentiellement s'exécuter en concurrence avec le premier qui est asynchrone. En effet, suivant la sémantique asynchrone (§15.2.3.1 p167) utilisée par le conteneur, les appels peuvent être séquentialisés (politique FIFO) ou non. Bien sûr, il faut que la politique soit FIFO dans notre exemple pour assurer un ordonnancement correct des écritures dans

le fichier. Par ailleurs, si la référence sur le *stored object* est diffusée, des requêtes concurrentes d'appel de méthode à distance peuvent être effectuées par des clients différents. La diffusion d'une référence distante impose de gérer la concurrence de ses accès (§10.2.4 p104). Si une politique asynchrone particulière doit être associée à un *stored object* à cet effet, une paire de références (§15.3 p172) doit être utilisée.

La récupération d'un proxy semi-transparent sur un *stored object* peut sembler fastidieuse. Heureusement, l'utilisateur final n'a pas à écrire tout ce code. En réalité, la classe `SORFactory` utilise en interne la classe `Instantiator` pour implémenter la méthode `newInstance()` de l'interface `Framework.Factory`. Aussi, les deux instructions :

```
1 Framework.setFactory(new SORFactory(activeMap));
2 jaya.java.io.FileWriter writer = new
3                         jaya.java.io.FileWriter(new File("/tmp/foo.txt"));
```

suffisent pour récupérer un proxy asynchrone semi-transparent sur un *stored object* instancié à distance.

Une remarque doit en outre être faite : l'appel à `jaya.mandala.jacob.Instantiator.instanciate()` a beau être synchrone, il n'en est pas moins distant et on peut se demander comment sont gérées les exceptions liées aux pannes réseaux (§10.2.3 p100). En fait, les *stored object*s sont manipulés par l'intermédiaire de leur conteneur actif associé. C'est donc dans le proxy distant de ce dernier que va se trouver la gestion de ces exceptions. On utilise un mécanisme événementiel illustré par le code suivant :

```
1  import java.lang.reflect.Method;
2  import mandala.util.ExceptionHandler;
3  import mandala.rami.FutureServer;
4  import mandala.jacob.remote.RemoteActiveMap;
5  import mandala.jacob.remote.AbstractRemote.ExceptionInfo;
6  ...
7      static final Method method instanciateMethod =
8          Instantiator.class.getMethod("instanciate", ...);
9      // Assume this instanciator is always "on" even if its related active map
10     // is inneficient
11     private final jaya.mandala.util.Instantiator fallbackInstanciator = ...
12 ...
13     // RemoteActiveMap implements the mandala.jacob.remote.Remote
14     // interface which provides the 'setExceptionHandler()' method.
15     RemoteActiveMap activeMap = ...; // Use JNDI to get a remote proxy
16     activeMap.setExceptionHandler(new ExceptionHandler() {
17         public Object handleException(Object o) {
18             // ExceptionHandler is not specific to remote aspect!
19             ExceptionInfo info = (ExceptionInfo) o;
```

```
20              // Singletons are provided so equals() is not necessary
21              if (info.target == activeMap &&
22                  info.method == callMethod) {
23                  // so it is ActiveMap.call(Object key, FutureServer f) which
24                  // throws an exception
25                  FutureServer f = (FutureServer) info.args[1];
26                  Method m = f.getMethod();
27                  if (m == instanciateMethod) {
28                      // When instanciator.instanciate() throws an exception use
29                      // fallbackInstanciator instead transparently
30                      Object[] args = f.getArgs();
31                      Constructor c = (Constructor) args[0];
32                      Object[] constructorArgs = (Object[]) args[1];
33
34                      return fallbackInstanciator.instanciate(c,
35                                                              constructorArgs);
36                  }
37              }
38
39              // In any other case, reject the handling of exceptions
40              return ExceptionHandler.RejectExceptionHandling.REJECT;
41          }
42          });
43  ...
```

Le principe est d'associer à tous les objets *primitifs distants* (§10.2.3.3 p102), tels que les conteneurs actifs, un gestionnaire d'exceptions capable de résoudre le problème de manière transparente pour l'appelant (en retournant un substitut du même type que l'appel distant original ayant levé l'exception). Dans notre cas, on suppose que l'on a un conteneur actif distant de secours toujours disponible. Les objets qui n'ont pu être instanciés à distance par un `instanciator` le sont dans le conteneur de secours en utilisant `fallbackInstanciator`. Pour l'appelant, l'invocation s'est bien déroulée[8]. Par contre, si l'exception ne concerne pas la méthode `instanciate()`, notre gestionnaire d'exception, en retournant `ExceptionHandler.RejectExceptionHandling.REJECT` indique au proxy distant que l'exception non-contrôlée `mandala.jacob.remote.TransportException` doit être levée dans le contexte de l'appelant.

[8]Dans une certaine mesure. L'appelant peut en effet se rendre compte que l'objet n'a pas été inséré dans le bon conteneur !

2.2.5 Séquences de caractères

Pour terminer notre série d'exemples, nous allons montrer les possibilités d'interfaçage de Mandala avec des bibliothèques existantes. Considérons le code suivant :

```
1  java.awt.image.renderable.RenderableImage
2  java.awt.image.RenderedImage
3  java.awt.image.Raster
4  ...
5  RenderableImage renderable = ...;
6  library.use(renderable);
7  RenderedImage rendered = renderable.createScaledRendering(width, heigth, ...);
8  Raster raster = rendered.getRaster();
9  ...
10 doSomethingElse();
```

Dans ce code, on suppose que la méthode `use()` est une méthode métier qui doit être utilisée (par exemple le code source de la bibliothèque n'est pas disponible). On suppose aussi que les opérations effectuées par cette méthode sur notre image sont relativement lentes. Il peut sembler opportun de rendre concurrent l'exécution de la méthode `renderable.createScaledRendering()` et de la méthode `doSomethingElse()`.

Le problème soulevé ici est lié à la nécessité d'utiliser cette méthode `use()` qui prend en paramètre un `RenderableImage`. Mandala propose un mécanisme totalement transparent (§16.1 p180) qui permet d'effectuer des appels asynchrones et éventuellement distants sans modification du code. Ainsi, le code précédent peut être rendu plus efficace en effectuant quelques modifications mineures :

```
1  java.awt.image.renderable.RenderableImage
2  java.awt.image.RenderedImage
3  java.awt.image.Raster
4  import mandala.rami.Framework;
5  ...
6  RenderableImage renderable = ...;
7  // Same type, but a total transparent asynchronous proxy
8  renderable = Framework.getTotalTransparentAsynchronousProxy(renderable);
9  // Asynchronous call, returns a future
10 RenderedImage rendered = renderable.createScaledRendering(width, heigth, ...);
11 // Use renderable asynchronously and transparently
12 library.use(renderable);
13 doSomethingElse();
14 Raster raster = rendered.getRaster();
```

Dans cet exemple, la variable `renderable` référence un proxy asynchrone totalement transparent ce qui permet de le passer à notre méthode `use()` sans qu'elle

s'aperçoive que les appels de méthodes sur cet objet sont asynchrones. Le mécanisme *d'attente par nécessité* (§12.4.1.1 p134) est utilisé ce qui assure que l'utilisation d'un objet retourné par un appel de méthode asynchrone totalement transparent est une opération bloquante. Ainsi, les méthodes `doSomethingElse()` et `createScaled-Rendering()` sont exécutées en concurrence sans que cela apparaisse dans le code explicitement. Si ce mécanisme semble élégant de prime abord, il pose de nombreux problèmes (§16.1.4 p185). Aussi Mandala impose de nombreuses contraintes dans l'utilisation de la transparence totale et on déconseillera son utilisation au profit de la semi-transparence (§16.2 p186).

2.3 Comparaisons

Le tableau Tab. 2.2 p22 dresse un comparatif de Mandala avec des travaux similaires, en donnant pour chaque élément la partie de ce document qui s'y réfère. Pour des raisons de place, quelques travaux étudiés dans ce document ne sont pas présentés. Les raisons de leur éviction sont les suivantes :

- les agents mobiles (§4 p41) sont de plus haut niveau et peuvent être implémentés à l'aide de Mandala comme nous le verrons en §20.1 p227 ;
- les processus légers mobiles (§5 p55) sont de plus bas niveau que notre plate-forme ;
- CentiJ (§9.2 p81) est relativement "basique" par rapport à notre plate-forme: c'est un générateur de *proxy* RMI ;
- Do! (§9.5 p86) impose un modèle de programmation parallèle qui n'est pas de type *appels de méthodes* ou même *passages de messages*.

Il existe un très grand nombre de travaux qui auraient pu être présents dans ce tableau. Toutefois, nous pensons que ce dernier donne un bon aperçu des possibilités de la plate-forme Mandala en la comparant aux travaux les plus représentatifs dans ce domaine.

Enfin, le tableau Tab. 2.3 p23 présente une comparaison d'un certain nombre de caractéristiques sur lesquelles nous avons travaillé avec les travaux similaires que nous avons étudiés. CentiJ n'est pas présenté dans ce tableau puisqu'il offre essentiellement *le dynamisme*. Pour le reste, il est comparable à RMI. Il est par ailleurs assez difficile de faire une comparaison avec des systèmes d'agents mobiles étant donné les différences qui existent entre ces systèmes, et surtout du fait que ces systèmes sont de beaucoup plus haut niveau que Mandala (et donc des autres travaux qui y sont comparables).

Thèmes de comparaison	RMI/Corba (§9.1 p79)	ProActive (§9.3 p82)	JavaParty (§9.4 p85)	EJB (§9.7 p87)	JavaSpace (§9.8 p88)	Mandala [212] (§V p199)
Caractéristique principale	Appel de méthode à distance	*Objets actifs* (§12.4.1.1 p134)	-Dédié au *clusters* de stations Multi-protocoles	Composants logiciels	Conteneurs distants	*Conteneurs actifs* (§8 p75) et *référence asynchrones* (§14 p147)
Avantages principaux	Standard de fait	-Dynamisme -Appels de méthodes asynchrones (distant) -Migration de code -Communications groupées typées	-Performance -Extension du langage Java au cas distant	Standard J2EE	-Composants faiblement couplés -API très simple -Requêtes transactionnelles -Recherche par *template*	-Dynamisme -Appels de méthodes asynchrones (distant) -Migration de code -Multi-protocoles (§10.3.2 p107) -Asynchronisme *complet* (§12.1 p123) -Semi-transparence (§16.2 p186)
Inconvénients principaux	-Statique -Performance médiocre -Réutilisation de code difficile -Gestion pénible des exceptions réseaux	-Transparence totale (§16.1 p180) -Critère d'activabilité (§12.4.1.1 p137) -Asynchronisme *côté serveur* seulement (§12.1 p123) -Proxy par héritage -Basé sur RMI	-Statique -Dédié au cas distant -Pas d'appel asynchrone -Contrainte de réseaux fiable -Extension du langage Java -Nombre de classes générées (débogage)	-Statique -Dédié au cas distant -Prise en main lourde -Réutilisation de code compliqué -Pas d'appel asynchrone -Basé sur RMI	-Dédié au cas distant -Passivité des objets stockés -Lié à JINI (lourd)	-Pas de compilateur depuis les sources (§16.2.3 p189) -Deux concepts à maîtriser: conteneurs actifs et références asynchrones

TAB. 2.2 – Tableau comparatif de Mandala et de travaux similaires

	RMI/Corba (§9.1 p79)	ProActive (§9.3 p82)	JavaParty (§9.4 p85)	Do! (§9.5 p86)	EJB (§9.7 p87)	JavaSpace (§9.8 p88)	Mandala [212] (§V p199)
Dynamisme (§10.5.1 p112)	Non	Oui	Non	Non	Non	Oui	Oui
Protocoles	JRMP et IIOP	RMI	Multi (KaRMI)	RMI	RMI	RMI	Multi (GRP[ab])
Gestion des pannes réseaux (§10.2.3 p100)	Exceptions contrôlées	Exceptions non-contrôlées	Supposées inexistantes	RMI	RMI	RMI	Evénementielles[c]
Appel asynchrone (§12.1 p123)	Non	Côté serveur (§12.1 p123)	Non	Non	Non	Non	-Côté serveur -Côté client -Complet (§15.3 p172) -Annulable (§14.5 p153)
Transparence	Semi[d]	Totale[e]	Totale	Semi[f]	RMI	RMI	Totale[e] et Semi (§16 p179)
Modèle local[g]	Oui[h]	Oui	Non	Oui	Non	Non	Oui
Politique concurrente	Aucune	Paramétrable	Aucune	Threads[i]	RMI	RMI	Paramétrable
Récupération de résultat (§14.4 p152)	Standard	-Attente par nécessité -Explicite	Standard	Aucune[j]	RMI	Aucune[k]	-Explicite (*future*) -Callback -Attente par nécessité

[a] *Generic Protocol Framework* présenté en §10.3.2 p107.
[b] Une autre alternative est proposée avec le Multi-Protocol Stored Object (MPSO) présenté en §9.11.2 p92
[c] Ce qui permet de fournir – entre-autres – les exceptions non-contrôlées §10.2.3.2 p102.
[d] Les méthodes doivent avoir une clause `throws RemoteException` dans leurs signatures.
[e] Ce qui pose un problème lorsque les appels sont asynchrones (*cf.* §16.1 p180).
[f] Il faut utiliser un modèle de programmation particulier.
[g] C'est cette caractéristique qui permet d'unifier la programmation locale et la programmation distribuée.
[h] Paradigme commun : appel de méthode, mais la sémantique des appels diffère (*cf.* §14.3.1 p149).
[i] Lorsque la méthode `Task.start()` est invoquée, une thread est créée.
[j] La méthode `Par.call()` retourne `void` lorsque toutes les `Task` ont terminé.
[k] Les objets stockés sont passifs dans JavaSpace !

Tab. 2.3 – Caractéristiques de Mandala.

Deuxième partie

Une approche de la mobilité

Chapitre 3

Les codes mobiles

Le domaine de recherche consacré aux codes mobiles est très vaste : à une extrémité nous distinguons la mobilité exploitée dans un contexte de calcul haute performance à des fins d'équilibrage de charge ou de tolérance aux pannes essentiellement (migration de processus lourds ou légers) ; de l'autre, les cartes à microprocesseur sont des vecteurs de codes mobiles puisque la mobilité est une caractéristique intrinsèque de la carte. Entre les deux, nous verrons les agents mobiles, des entités qui empruntent l'autonomie au domaine de l'intelligence artificielle et la mobilité au domaine des systèmes distribués. Si l'ensemble de ces travaux utilise la caracatéristique mobile, on ne trouve pas dans chacun d'eux la même définition de la notion de mobilité. Néanmoins, il est globalement admis qu'un code est *mobile* s'il a la *capacité de reconfigurer dynamiquement le lien entre les composants logiciels de l'application et l'emplacement physique de leur exécution* [76].

Le concept de code mobile n'est pas nouveau. Toutefois, la fonctionnalité "mobile" a longtemps été introduite au niveau du système d'exploitation des machines parallèles. Ainsi, le développeur n'a ni le contrôle, ni la visibilité du processus de migration. Si ces techniques sont suffisantes pour un petit système distribué (*cluster* par exemple), elles ne sont pas adaptées aux réseaux à grande échelle tel qu'un grand Intranet ou même Internet. On souhaite en effet utiliser la migration de code dans des buts très variés tels que l'ajout dynamique de services à un serveur, le déploiement d'applications, la tolérance aux pannes et l'utilisation de liaisons à très bas débits ou à faible connectivité (RTC, réseaux sans fil).

Par exemple, la fonction mobile peut être utilisée par l'utilisateur A d'une liaison RTC – dont le coût dépend de la durée de connexion – qui souhaite télécharger un gros document disponible seulement chez un autre utilisateur B relié lui aussi par une liaison RTC (par exemple, un album de photos de famille). Du point de vue de l'utilisateur final (celui qui télécharge) il est souhaitable d'obtenir les garanties suivantes :

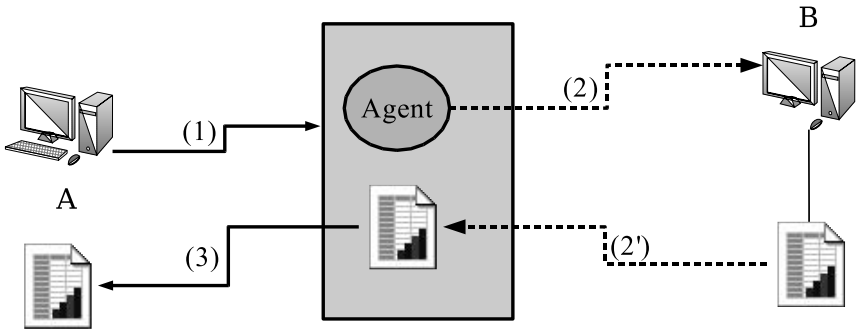


FIG. 3.1 – Téléchargement opportuniste.

- téléchargement au débit maximum pour minimiser le temps de connexion et donc le coût du transfert ;
- téléchargement en une fois : l'autre utilisateur étant lui aussi sur une liaison RTC, il n'est pas raisonnable d'attendre sa connexion pour commencer ou poursuivre le téléchargement.

Les outils de téléchargement classique (FTP, HTTP) n'offrent pas ces garanties. Il faut une solution *opportuniste* de téléchargement illustré par la figure FIG. 3.1 p28 :

1. on suppose qu'un serveur d'accueil de code mobile est disponible sur Internet en permanence (un fournisseur de service par exemple) et offre des débits importants ;
2. l'utilisateur A envoie un code mobile sur ce serveur dont le rôle est de télécharger le document (1) ;
3. l'utilisateur A se déconnecte (s'il le souhaite) ;
4. le code sur le serveur tente de se connecter chez l'utilisateur B (2), détenteur du document ; à chaque déconnexion de B, le code mobile garde la partie téléchargée et tente de se reconnecter afin de reprendre le téléchargement (2') au point de coupure et cela jusqu'au téléchargement complet du document ;
5. lorsque le téléchargement est terminé, le code tente d'en avertir A ; il attend une connexion de A ;
6. lorsque l'utilisateur A se reconnecte, et que le document a été complètement téléchargé par le code mobile, il reçoit un avertissement ;
7. l'utilisateur A peut alors récupérer le document (3) quand il le désire, en une fois, et au débit maximum[1].

[1]En fait, le débit ne peut être garanti que par le fournisseur de service. Toutefois, dans le cadre d'une liaison RTC, on peut considérer que c'est le cas (très faible débit).

Évidemment, l'utilisation d'un code mobile n'est pas nécessairement justifiée ici. Il est parfaitement envisageable d'utiliser un serveur dédié à la fonction de téléchargement. Dans ce cas, l'utilisateur A fournit simplement une requête de téléchargement du document chez l'utilisateur B (par le biais d'une URL par exemple) et le serveur s'occupe du téléchargement et de l'avertissement. Cependant, cette solution requiert l'installation préalable du service de téléchargement. On imagine bien que la maintenance d'une telle architecture dans laquelle chaque nouveau service doit être préalablement installé et configuré, est sans commune mesure avec celle qui consiste à configurer un serveur de codes mobiles.

Pour le concepteur de l'application, l'aspect mobile est une propriété fonctionnelle. Le support pour la mobilité de code doit donc lui permettre de contrôler la localisation de son code : l'aspect mobile doit apparaître au niveau du langage (ou par le biais d'une bibliothèque) et ne doit pas être masqué.

3.1 Terminologie

En reprenant la terminologie introduite en [43], la localisation dans un système de code mobile, est représentée par le *support d'exécution* (SE) qui accueille des *ressources* (R) et des *unités d'exécutions* (UE). Une ressource est une entité qui peut être partagée par différentes unités d'exécution comme un fichier ou un objet. L'unité d'exécution représente un flot séquentiel de calcul (*thread*) et se compose d'un *code* et d'un *contexte* comme l'illustre la figure Fig. 3.2 p30. Le contexte contient l'*espace des données* – l'ensemble des références vers les ressources auxquelles accède l'unité d'exécution – et l'*état d'exécution* – qui contient l'ensemble des données non partagées par l'unité d'exécution ainsi que les informations de contrôle telles que la pile d'exécution et le pointeur d'instruction.

Dans ce cadre, on définit deux types de migration :

- **la migration forte :** c'est la capacité du système à déplacer à la fois le code et le contexte des entités d'exécution ;
- **la migration faible :** c'est la capacité du système à déplacer le code des entités d'exécution, mais pas leur contexte. Des données d'initialisation peuvent néanmoins être transférées avec le code. Généralement, après une migration faible, un mécanisme particulier est mis en place pour restaurer un contexte cohérent en utilisant les données d'initialisation (utilisation d'un *callback* par exemple).

On qualifie enfin une migration de *proactive* lorsque le choix de la date et de la destination de la migration sont réalisés de manière autonome par l'unité d'exécution

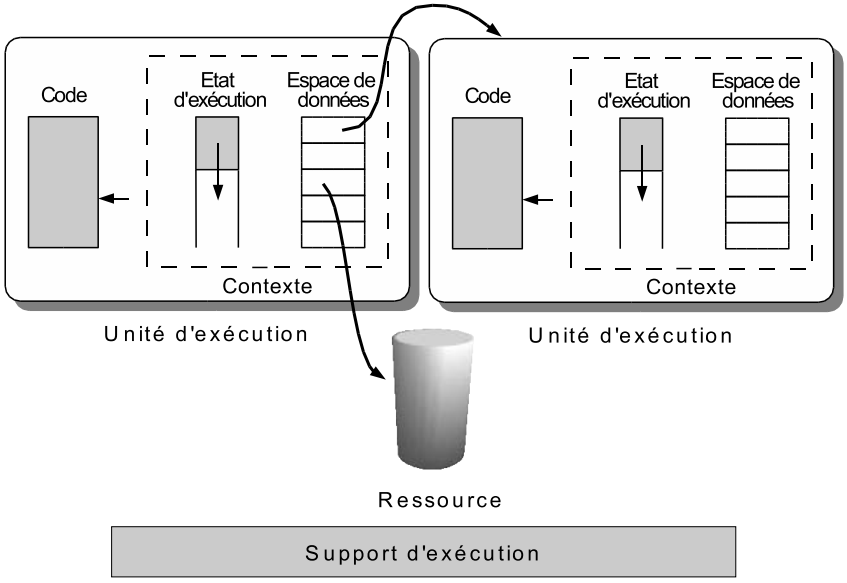


FIG. 3.2 – Les différents composants d'un système de codes mobiles.

migrante. Une migration est *réactive*, lorsque l'ordre est donné par un tiers (système, ou autre unité d'exécution).

3.2 Gestion de l'espace des données

Suite à la migration d'une unité d'exécution, l'espace de données peut se trouver dans un état incohérent.

Par exemple, la migration d'une unité d'exécution qui référence une structure de données par l'intermédiaire d'un pointeur en langage C, doit mettre en œuvre un mécanisme lors de l'arrivée à destination qui permette à l'unité d'exécution de retrouver à l'adresse du pointeur la structure de données telle qu'elle était avant de migrer (déplacement de données et mécanisme d'isoadressage[2] par exemple).

Évidemment, dans le cas des machines à mémoire partagée, la gestion de l'espace des données ne pose guère de problème puisque le système donne l'illusion d'une machine virtuelle unique. Toutefois, dans ce cas, la migration est généralement une fonctionnalité système, qui n'est pas accessible au programmeur, la notion de machine destination n'existant pas.

Le système peut donc proposer des mécanismes de *gestion de l'espace des données* [76] pour replacer l'unité d'exécution dans un contexte cohérent. Cependant, ceci est délicat à mettre en œuvre pour des raisons techniques et sémantiques. Non

[2]Un mécanisme d'isoadressage permet d'assigner des adresses virtuelles globalement unique dans un système à mémoire distribuée.

seulement, il faut implémenter différents mécanismes (copie, déplacement, référencement à distance) en fonction des ressources référencées dans l'espace des données (fichier, socket, verrou, donnée partagée, console, etc.) d'une unité d'exécution, mais surtout, le mécanisme à utiliser ne peut être connu qu'à l'exécution par l'unité d'exécution en fonction de son utilisation de la ressource. Par exemple lorsqu'une unité d'exécution manipule un fichier, son espace de données contient une référence vers lui (descripteur de fichier POSIX). Lorsque l'unité d'exécution migre sur une autre machine, la référence n'est probablement pas valide. Si le fichier est de petite taille et que l'unité d'exécution a besoin de l'ensemble du contenu, la copie sera probablement plus efficace qu'une référence distante. Par contre, si le fichier est un gros fichier et que l'unité d'exécution n'a besoin d'en lire qu'une partie, une référence distante sera sans doute plus appropriée. Enfin, si le fichier est un fichier temporaire uniquement créé pour les besoins de l'unité d'exécution, son déplacement s'avère vraisemblablement plus judicieux.

Enfin, nous devons distinguer la gestion de l'espace des données dans les cas de migration forte ou faible. L'intérêt de la migration forte est d'affranchir le programmeur de toute contrainte liée à la mobilité : le code continue son exécution sur une machine distante de manière transparente. Cela impose donc au système de gérer automatiquement l'espace des données d'une entité d'exécution. Or nous l'avons vu, la gestion automatique est très complexe à réaliser. Dans le cas d'une migration faible, le développeur doit participer à la gestion de l'espace des données en utilisant des données d'initialisation qui sont transférées avec le code. Le programmeur doit lui-même fournir ces données qui permettront lors de l'initialisation de son code après sa migration, de le placer dans un état cohérent. Ce genre de manipulation peut s'avérer très coûteux en temps de développement, temps de débogage et temps d'exécution. D'un autre côté, c'est le développeur qui choisit la meilleure solution pour restaurer l'espace des données.

Enfin, selon que la migration est réactive ou proactive, la gestion de l'espace des données ne peut être réalisée de la même manière. Dans le cas d'une migration proactive, le développeur spécifie lui-même dans le code les points de migration. Aussi, il est capable de préparer la migration de façon à retrouver un espace des données cohérent. Il peut par exemple, relâcher un verrou sur une structure de données partagée en lecture/écriture par plusieurs flots d'exécution (threads ou processus) avant sa migration. Cela n'est pas possible avec une migration réactive puisque la migration peut survenir à n'importe quel point du code. Généralement un mécanisme permet aux code d'indiquer qu'il n'est pas dans un état *migrable* ce qui informe le support d'exécution que ce code ne peut être déplacé. Nous verrons que ce problème de gestion de l'espace des données revient régulièrement.

3.3 Paradigme de conception

Les applications distribuées si elles s'appuient sur une technologie particulière doivent avant tout être définies suivant une architecture logicielle adaptée. Une architecture logicielle est décrite par [43] :

- **les composants de *code* (*code components*)** représentent le *savoir-faire* utilisé pour effectuer un calcul ;
- **les composants *ressources* (*ressource components*)** représentent les données et les périphériques utilisés pendant le calcul ;
- **les composants de *calculs*** représentent les exécutants qui effectuent le calcul spécifié par un composant de code ;
- **une *interaction*** représente un évènement qui fait intervenir au moins deux composants ;
- **un *site*** représente intuitivement l'emplacement d'un composant : une interaction entre deux composants localisés sur des sites distincts est considérée moins efficace qu'une interaction entre deux composants co-localisés sur le même site.

L'exécution d'un calcul ne peut avoir lieu que si le code qui décrit le calcul, les ressources utilisées par le calcul et les composants de calcul responsables de l'exécution du calcul sont co-localisés sur *le même site*.

Les architectures logicielles utilisées dans les applications distribuées qui ont des caractéristiques semblables sont regroupées en *paradigmes de conception*. Les paradigmes de conception sont décrits par des modèles d'interactions qui définissent les déplacements des composants et les coordinations entre composants nécessaires à l'accomplissement du service. Ces paradigmes sont caractérisés par l'emplacement des composants avant et après l'exécution d'un service, par le composant de calcul responsable de l'exécution du code et par l'emplacement où le calcul effectif a eu lieu.

Considérons un scénario dans lequel un composant de calcul A localisé sur un site S_A souhaite obtenir le résultat d'un service. On suppose qu'il existe sur un site S_B, un composant de calcul B qui sera impliqué dans l'exécution de ce service. Le paradigme client/serveur est le plus connu et le plus utilisé des paradigmes de conception d'applications distribuées. Trois autres paradigmes de conception exploitant la mobilité de code peuvent être identifiés : le paradigme d'évaluation à distance, le paradigme code à la demande et le paradigme agent mobile. La table TAB. 3.1 p33 montre les différences fondamentales entre ces différents paradigmes en terme de localisation des composants de calculs et en déplacement de composants.

Paradigme	Avant		Après	
	S_A	S_B	S_A	S_B
Client-Serveur	A	code ressource B	A	code ressource **B**
Évaluation à distance	code A	ressource B	A	*code* ressource **B**
Code à la demande	ressource A	code B	*code* ressource **A**	B
Agent mobile	code A	ressource	–	*code* ressource ***A***

Tab. 3.1 – Cette table montre la localisation des composants avant et après l'exécution du service. Pour chaque paradigme, le composant de calcul en **gras** est celui qui exécute le code. Les composants en *italiques* sont ceux qui ont été déplacés.

3.3.1 Le paradigme client-serveur

Dans l'approche *client/serveur*, un composant logiciel A (le client) localisé sur un site S_A demande à un composant de calcul B (le serveur) localisé sur un site S_B d'exécuter un service par l'intermédiaire d'une interaction. Le code et les ressources sont aussi co-localisés sur le site S_B. B exécute le service en impliquant le composant de code correspondant au service et en utilisant les ressources nécessaires co-localisées avec B. En général, l'exécution du service produit une réponse qui est renvoyée au composant A par l'intermédiaire d'une nouvelle interaction. Bien que ce paradigme n'utilise pas la notion de migration nous le présentons ici pour l'opposer à ceux qui, justement, exploitent la mobilité de code.

Le paradigme client/serveur est probablement celui qui est le plus utilisé aujourd'hui. Parmi les exemples bien connus, nous pouvons citer X-Windows, HTTP, SMTP, FTP, NFS, NIS, et LDAP.

Parfois, l'exécution du service peut nécessiter une ou plusieurs interactions supplémentaires avec d'autres composants (architecture multi-tiers). Dans ce cas, le serveur devient le client d'une nouvelle interaction avec un autre composant de calcul. Du point de vue de A, le serveur contient l'ensemble des ressources – au sens général du terme – pour l'exécution du service demandé.

3.3.2 Le paradigme évaluation à distance

Dans l'approche *évaluation à distance*, un composant A possède le code requis pour l'exécution d'un service mais ne possède pas les ressources adéquates qui sont localisées sur un site S_B. Aussi, A transmet le code à un composant de calcul B qui utilise ses ressources locales pour l'exécution du service.

L'évaluation à distance est utilisée depuis fort longtemps. Par exemple, la commande `rsh` – bien connue du monde UNIX – permet à l'utilisateur d'exécuter un script sur la machine spécifiée en argument. Dans cet exemple, A est le processus `rsh`, B le démon `rshd`, les ressources sont celles disponibles chez B, le code est le script qui est passé en paramètre à `rsh`.

L'impression d'un document sur une imprimante PostScript[103] est un autre exemple de l'utilisation de ce paradigme. Sans rentrer dans la jungle des gestionnaires d'impressions (LPR, LPR-NG, CUPS, etc.) et de leur architecture logicielle, on peut considérer que le code est le document PostScript lui-même et que le composant de calcul est l'imprimante qui contient l'ensemble des ressources (papier, encre, ...).

3.3.3 Le paradigme code à la demande

Dans l'approche *code à la demande*, un composant A et l'ensemble des ressources nécessaires à l'exécution d'un service sont co-localisés sur un site S_A. Par contre, A ne connaît pas le code – le savoir-faire – qui lui permettrait d'exécuter le service. Ce code étant disponible sur un site S_B, A interagit avec un composant B sur S_B en demandant le code du service. B retourne le code du service demandé par une seconde interaction.

Une grande partie des applications multimédia disponibles sur Internet fonctionnent selon ce principe. Le navigateur récupère un document qu'il ne sait pas interpréter. Un en-tête dans ce document permet de localiser le composant logiciel capable d'interpréter ce document. Après avoir récupéré ce composant, le navigateur est en mesure d'exploiter le document. Ainsi fonctionnent les applets Java, mais aussi Macromedia® Flash™, etc.

3.3.4 Le paradigme agent mobile

Dans l'approche *agent mobile*, un composant A sur un site S_A détient un code correspondant à un service. Une partie des ressources nécessaires à l'exécution du service est localisée sur un site S_B. Aussi, A migre sur le site S_B, en emportant éventuellement avec lui des résultats intermédiaires. Une fois sur le site S_B, A termine l'exécution du service en exploitant les ressources disponible sur S_B. Il est important de faire la distinction entre le paradigme agent mobile et les autres paradigmes. En

effet, dans cette approche, la migration implique le transfert d'un composant entier en cours d'exécution comprenant son code mais aussi son contexte et un certain nombre de ressources.

Une étude plus complète de ce paradigme sera effectuée en §4 p41.

3.4 Langages pour codes mobiles

Un certain nombre d'études [113, 57] ont montré que l'aspect mobile pose de nouvelles contraintes qui doivent être prises en compte par le langage utilisé pour le développement d'applications exploitant la migration de code.

3.4.1 Système de typage

Le système de typage utilisé dans un langage est un élément essentiel, une caractéristique fondamentale d'un langage donné. Deux familles peuvent être considérées :

- **les langages *non-typés* :** le type des données est fonction de l'opérateur utilisé. La puissance d'expression de ces langages est contrebalancée par la difficulté de protéger les données de mauvaises manipulations sémantiques.
- **les langages fortement *typés* :** l'absence d'erreur liée au typage peut être vérifiée statiquement. Toutefois, le système de typage est souvent *affaibli* – pour des raisons de souplesse – par l'utilisation d'un type générique. Le système peut alors fournir un support d'exécution à même de vérifier que les types sont manipulés au travers d'opérations légales. Cela complique donc dans une certaine mesure la vérification de programme.

Les langages pour le code mobile ne peuvent pratiquement pas être fortement typés puisque d'une part, le code mobile ne peut pas – par définition – être statiquement typé, et d'autre part, lorsqu'un code arrive à destination, les ressources de l'hôte – non nécessairement connues au moment de la conception du code – doivent lui être associées. Par conséquent, les langages utilisés dans un contexte mobile sont soit non-typés, soit faiblement typés par l'intermédiaire d'un type générique. C'est le cas du langage Java qui utilise la classe `Object` à cet effet. Par ailleurs, en Java, l'utilisation de la réflexion affaiblit encore le typage, au point que nous serons amené à fournir des solutions en §16 p179.

3.4.2 Résolution de noms

La résolution des noms est un problème majeur dans les langages pour code mobile. En effet, dans l'écriture d'un programme exploitant la migration de code,

les identificateurs utilisés peuvent être liés à des entités distantes. La résolution des noms est directement liée à la *gestion de l'espace des données* du support d'exécution que nous avons décrit en §3.2 p30. Nous avons vu que cette gestion dépend du type de migration (forte ou faible, proactive ou réactive). Dans le cas d'une gestion automatique de l'espace des données, le langage n'a pas à faire apparaître cette problématique : le support d'exécution s'occupe de tout. Dans le cas contraire, le langage doit faciliter la tâche du programmeur en lui permettant :

- d'exprimer que son code n'est pas dans un état migrable (cas de migration réactive) ;
- d'utiliser des mécanismes de copie, de déplacement, ou de référencement à distance.

En utilisant des mécanismes événementiels, le développeur peut être en mesure de régler ce problème. Lors d'une migration, il prépare son code en sauvegardant les informations nécessaires à la résolution des noms : référencement à distance, copie ou déplacement. Le langage doit faciliter l'implémentation de ces mécanismes.

3.4.3 Liaison dynamique

Les mécanismes de liaison dynamique dans le cas local – par l'utilisation de bibliothèques liées dynamiquements (*dynamically linked libraries* ou DLL) – sont naturellement étendus au domaine de la migration de code. Deux cas se présentent:

1. le support d'exécution doit accueillir une unité d'exécution inconnue: un mécanisme de téléchargement doit être mis en place (code à la demande) et la liaison de ce nouveau code avec le code du support d'exécution doit être effectuée;
2. l'unité d'exécution arrive à destination d'un support d'exécution et les ressources de ce dernier doivent être liées à la première (fichiers et bibliothèques par exemple).

Les langages interprétés se prêtent généralement mieux aux mécanismes de liaison dynamique. Le *runtime* Java[3] par exemple contient une classe particulière pour le chargement dynamique de classes (`ClassLoader`). Les mécanismes de réflexions [148] sont aussi très utiles pour faciliter la liaison d'un code inconnu au sein d'une application.

3.4.4 Mode d'exécution

Le choix de l'exécution d'un langage de programmation par interprétation directe ou au travers d'une phase de compilation ne peut être fait sans considérer les

[3]Le *runtime* Java n'est pas seulement réduit à la machine virtuelle, mais contient aussi l'ensemble des classes des paquetages du JDK.

contraintes imposées par la migration de code. Deux contraintes essentielles vont guider ce choix :

- **la portabilité :** nous avons vu au chapitre §3 p27 que l'objectif de la migration de code est d'être utilisé dans un cadre large et hétérogène comme Internet[4] ;
- **la sécurité :** le code qui s'exécute sur un support d'exécution ne doit pas mettre en péril les ressources du système.

Le mode d'exécution interprété offre une grande portabilité et permet d'effectuer des vérifications au chargement et à l'exécution du code. Enfin, l'absence d'arithmétique sur les pointeurs apporte un certain nombre de garanties liées à la sécurité (accès à des champs privés en manipulant directement la mémoire par exemple). Java possède cette caractéristique et utilise un ramasse-miettes pour s'assurer de la destruction des objets non-référencés.

3.4.5 Conclusion

Nous avons vu les contraintes qu'imposent la migration de code au niveau du langage. Java nous est imposé par le système JEM et la forte implication de l'équipe dans ce langage [21], il possède les caractéristiques requises pour l'implémentation d'un système de code mobile [59]. En outre, la sérialisation intégrée au langage facilite grandement la réalisation du mécanisme de migration de code.

Java est donc très utilisé en raison de son adéquation avec le concept de migration de code [58].

3.5 Sécurité

L'enjeu majeur d'un système à migration de code est la gestion de la sécurité. Permettre à un support d'exécution de recevoir et d'exécuter une unité d'exécution dont l'origine n'est pas connue pose des problèmes évidents. Deux types de communications sont à considérer impliquant deux types de sécurité différents comme illustré par la figure Fig. 3.3 p38 :

- la sécurité *inter-support d'exécution* (ou sécurité *externe*) qui passe par des mécanismes d'authentification mutuelle entre une unité d'exécution migrante et le support d'exécution destination mais aussi par la confidentialité et l'intégrité des communications entre supports d'exécution ;

[4] Le terme *transportabilité* serait sans doute plus approprié ici : un code C ANSI peut être écrit de façon portable mais une phase de compilation reste nécessaire pour l'exécuter sur une autre architecture processeur ce qui n'est pas le cas d'un code *transportable* tel qu'une classe Java ou un shell script. Nous ne ferons plus cette distinction dans la suite.

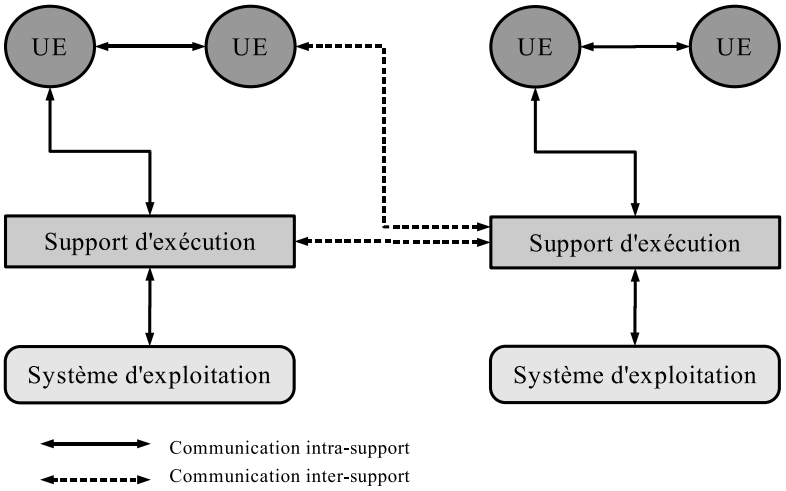


FIG. 3.3 – Types de sécurité dans le code mobile.

- la sécurité *intra-support d'exécution* (ou sécurité *interne*) qui comprend des mécanismes de protection entre unités d'exécution, entre les unités d'exécution et leur support d'exécution et entre le support d'exécution et le système d'exploitation.

Si le type *inter-support* est "facilement" mis en œuvre en utilisant des mécanismes d'authentification et de cryptage bien connus, il n'en va pas de même pour la sécurité *intra-support*. En effet, s'il existe des outils pour contractualiser les ressources utilisées par des composants logiciels [122] elles engendrent un certain coût qui peut être prohibitif au vu des fonctionnalités recherchées. Dans ce cas, on préférera se passer de la migration de code et utiliser un paradigme client/serveur "classique" [95].

Finalement, la sécurité la plus complexe à garantir est la protection des unités d'exécution du support d'exécution lui-même. En effet, puisque le *support doit exécuter l'unité*, il a accès à la représentation interne de celle-ci.

Ce problème n'est d'ailleurs généralement pas étudié dans les systèmes de code mobiles classiques, et les supports d'exécution sont généralement considérés comme "de confiance". Nous proposons une solution en §6.3.3 p69 dédiée aux cartes à microprocesseur.

Enfin, les mécanismes de sécurité utilisés dans le cadre des systèmes à migration de code peuvent être accessibles au programmeur. Cela offre une grande flexibilité au dépend d'une complexification du langage et donc de l'applicatif.

Dans ce cadre, Java intégre plusieurs mécanismes de sécurité à plusieurs niveaux :

- le bytecode des classes chargées dans la machine virtuelle est vérifié ;
- la classe `SecurityManager` peut être utilisée pour définir des politiques de sécurité ;
- des bibliothèques de plus haut niveau facilitent l'implémentation de mécanismes d'authentification et de cryptage.

3.6 Conclusion

Nous avons donné la terminologie employée dans le domaine de la migration de code, étudié les trois principaux paradigmes de conception qui délimitent le domaine, examiné les caractéristiques que doit avoir un langage de programmation de code mobile et analysé les problèmes liés à la sécurité. Nous allons maintenant nous pencher sur trois approches radicalement différentes qui exploitent chacune à leur manière la migration de code :

- les agents mobiles ;
- les processus légers mobiles ;
- les cartes à micro-processeurs.

Chapitre 4

Les agents mobiles

Notre recherche d'une entité d'exécution pour la plate-forme d'exécution JEM nous a conduit à étudier plus particulièrement les systèmes d'agents mobiles écrits en Java.

4.1 Terminologie

Une certaine confusion existe au niveau de la terminologie utilisée dans le domaine des agents mobiles. En effet, le terme "agent" est aussi employé dans le domaine de l'intelligence artificielle. Un agent est alors considéré comme "intelligent" s'il est capable d'effectuer des actions, de réagir à des évènements extérieurs de manière *autonome*. Un tel agent *apprend* à réagir en fonction de l'environnement extérieur ce qui induit une certaine latence avant l'obtention de réactions pertinentes.

Dans le domaine du distribué, un agent mobile n'est pas nécessairement intelligent.

Si la frontière entre ces deux domaines n'est pas nette, elle est néanmoins parfaitement spécifiée par deux standards distincts :

- **le domaine des agents mobiles** est spécifié par le document de l'OMG : *Mobile Agent System Interoperability Facilities Specification* (MASIF)[86] ;
- **le domaine des agents intelligents** est normalisé par la *Foundation for Intelligent Physical Agents* (FIPA) [6] qui définit un grand nombre de standards dans ce domaine.

Remarquons toutefois que dans la littérature on retrouve régulièrement la définition suivante (ou une équivalente) que nous adopterons aussi :

Définition 4.1.1 (Agent)
Un agent est un programme autonome *et* indépendant *:*

- **autonome :** *il a le contrôle de sa propre exécution et ne requiert donc pas d'interaction pour accomplir sa tâche*[1] *;*
- **indépendant :** *il possède ses propres threads d'exécution.*

La mobilité vient se greffer au dessus de cette définition par la faculté de se déplacer (migration forte ou faible, peu importe) d'un support d'exécution à un autre. Nécessairement, la migration d'un agent mobile est *proactive* sinon, l'agent n'est plus autonome et ce n'est plus un agent !

4.2 Intérêt

On peut se demander quel est l'intérêt d'utiliser le paradigme agents mobiles. Pourquoi les agents mobiles ? A quoi cela sert-il ? Si l'on peut trouver de multiples bonnes raisons à l'utilisation des agents mobiles [118, 60], on peut remarquer [95] que la plupart des applications peuvent être écrites – souvent plus simplement – avec des paradigmes plus classiques.

L'intérêt de ce paradigme ne peut être trouvé pour une application particulière mais pour un ensemble d'applications qui peuvent être implémentées à l'aide du même paradigme, simplifiant le développement (moins de technologies à apprendre, à maintenir, à contrôler) et le coût (formation). L'article "les agents mobiles et l'avenir de l'Internet" [116] est très clair à ce propos : il n'y a pas de *"killer application"* dans le domaine des agents mobiles. C'est donc une des raisons qui explique que des systèmes d'agents mobiles ne sont pas (encore) disponibles sur Internet.

D'autres raisons à cette frilosité doivent toutefois être prises en compte. Nous les détaillons ci-dessous.

4.2.1 Raisons techniques

4.2.1.1 Sécurité

L'accueil d'un agent sur son (ses) serveur(s) d'agents est une fonctionnalité qui comporte un risque maximal si la sécurité n'est pas prise en compte. Un agent peut en effet créer des agents sur le réseau et contaminer ainsi d'autres serveurs[2], polluer le disque, la mémoire, le processeur, etc. Toutefois, ce genre de problème a déjà été rencontré dans le domaine général des systèmes distribués et des mécanismes d'authentification par certificats, et des isolats (*sandbox*) permettent de les résoudre.

[1] En réalité, l'agent peut interagir avec d'autres entités logicielles telles un autre agent, ou une base de données, mais il ne requiert pas d'interaction avec son utilisateur.

[2] Comportement de type *vers*.

Ce n'est pas tout à fait la même chose en ce qui concerne la protection d'un agent contre un hôte mal intentionné. Par exemple, l'utilisateur d'un agent ne souhaite pas voir son agent modifié par un serveur d'agents. Si ces problèmes on déjà été étudiés et si des solutions sont théoriquement déjà disponibles [34] dans certains cas particuliers, il semble que dans le cas général des agents mobiles sur un réseau fortement hétérogène comme Internet, il n'y ait pas de solution [130]. Aussi, en utilisant un modèle abstrait et en s'appuyant conjointement sur des techniques d'analyses statiques et dynamiques de sécurité, on peut déterminer le degré de sécurité d'un système d'agents mobiles dans certains cas [163].

4.2.1.2 Interopérabilité

Il existe plusieurs systèmes d'agents incompatibles entre-eux qui diffèrent par :
- le type de migration supporté (forte ou faible, proactive ou réactive, ...) ;
- le langage utilisé (et donc la représentation interne de l'agent) ;
- la gestion de la sécurité ;
- la gestion de l'espace des données.

Ces problèmes d'interopérabilité sont complexes à résoudre comme en témoigne la difficulté qu'a eu le standard de l'OMG, MASIF[86], à émerger [223, 201, 87, 202, 15] et le peu de système d'agents qui s'y conforment (seul Grasshopper [37] – l'implémentation de référence que nous verrons en §4.3.3 p50 – y est conforme à 100%).

Dans un tel contexte, les utilisateurs potentiels d'un système d'agents mobiles doivent choisir parmi la liste de systèmes d'agents disponible sur le web [152].

Enfin, certain systèmes d'agents ne sont plus maintenus (AgentSpace [61]), ont été remplacés (Telescript [132] par Odyssey [85]), ou ne sont même plus disponibles (Concordia [151]). On comprend dès lors la réserve que peuvent avoir les administrateurs de systèmes censés installer et maintenir un ou plusieurs serveurs d'agents.

4.2.2 Raisons non-techniques

4.2.2.1 Mode de rémunération

Beaucoup de sites fonctionnent avec un *business model* basé sur la publicité. Plus il y a d'utilisateurs sur leur site, plus la rémunération est importante[3]. C'est le cas des sites *portails* (Yahoo!, AOL, MSN, etc.) qui offrent à l'internaute une information filtrée et mise à jour régulièrement. Si des agents sont utilisés à grande échelle, il y a moins "d'utilisateurs physiques" qui visitent leur site. Le mode de rémunération de ces entreprises doit être repensé. Ce n'est vraisemblablement pas

[3]Bien entendu, nous simplifions à l'extrême ici.

dans leur intérêt aujourd'hui. Or, l'adoption par l'un de ces sites d'un système ouvert d'agents mobiles donnerait un élan non négligeable à cette technologie.

4.2.2.2 Gestion de la qualité de service

Un fournisseur de services qui offrirait l'accès ouvert à un serveur d'agents serait confronté à un problème de qualité de service. En effet, n'ayant pas forcément la main[4] sur tous les agents qui visitent son serveur et utilisent les services qui y sont disponibles, le fournisseur ne peut contrôler *l'image* que percevra un utilisateur de ses services par l'intermédiaire de son agent. Si l'agent est mal implémenté, les services pourront être considérés par l'utilisateur final comme mauvais.

Remarquons que c'est déjà le cas pour le Web : la qualité des pages envoyées dépend aussi du navigateur utilisé. Il est ainsi courant de voir un site web critiqué parce qu'il ne permet pas un affichage correct de ces pages sur tous les navigateurs du marché.

4.2.2.3 Performance

Le problème avec les agents mobiles est d'étudier la performance d'un algorithme basé sur ce paradigme par rapport à d'autres. Parfois, l'intuition est trompeuse comme le montre l'étude [43] où l'utilisation du paradigme "agents mobiles" donne de moins bonnes performances que le paradigme "code à la demande" ou "évaluation à distance" pour un algorithme de monitoring distribué (à la SNMP).

Ce problème d'analyse du coût freine l'adoption de ce paradigme à grande échelle car on préfère un paradigme peut être moins "à la mode" mais dont les coûts sont parfaitement mesurables.

4.2.2.4 Maturité de la technologie

Enfin, l'ensemble des raisons précédentes peuvent être la conséquence d'un manque de maturité du paradigme agents mobiles. On connaît le décalage qu'il y a entre l'émergence d'une technologie, son adoption dans la communauté de la recherche et enfin son utilisation à grande échelle dans les entreprises et chez le particulier.

4.3 État de l'art

Malgré la relative jeunesse du domaine des agents mobiles, il existe un très grand nombre d'implémentations [152]. Il n'est pas raisonnable d'étudier de manière exhaustive l'ensemble de ces systèmes d'agents pour deux raisons : d'une part, un

[4]Développement de l'agent par un tiers.

certain nombre de fonctionnalités se retrouve régulièrement dans chacun d'entre-eux ce qui rendrait la lecture un peu répétitive ; d'autre part, et surtout, de nouveaux systèmes apparaissent régulièrement tandis que d'autres ne sont plus disponibles ou plus maintenus. Enfin, un certain nombre de systèmes d'agents – et non des moindres (nous en verrons trois sur quatre) – possèdent des licences commerciales ce qui complique leur téléchargement et donc leur étude. Toutefois, à ma connaissance, seul AgentX [4] est breveté.

Aussi nous nous limiterons à l'étude des systèmes d'agents mobiles écrits en Java[5]. Parmi ces systèmes, nous en verrons quatre qui nous semblent être les plus significatifs. Remarquons qu'une telle entreprise a été réalisée en [58] et que deux des systèmes étudiés ne sont déjà plus disponibles (Odyssey et Concordia).

4.3.1 AgentOS

AgentOS [115] est aux agents ce qu'UNIX est aux processus : une sorte de système d'exploitation, conçu à base d'agents au dessus de la machine virtuelle Java, pour la gestion des agents, par des agents.

AgentOS définit le cycle de vie d'un agent schématisé sur la figure Fig. 4.1 p46. Lors de la création d'un agent – sur un hôte ou un client AgentOS, un identifiant global unique lui est associé. L'agent est ensuite migré sur un système AgentOS, et peut donc commencer son travail. L'exécution d'un agent peut impliquer l'invocation de services disponibles sur le système ou la communication avec d'autres agents. Lorsque sa tâche est terminée, l'agent peut être soit détruit soit rendu inactif – par endormissement – jusqu'à une requête de récupération par la source. L'activité de l'agent peut l'amener à effectuer une requête de migration ce qui répète le cycle une nouvelle fois.

L'architecture d'AgentOS repose sur trois principales entités de base[6] :

- **le noyau :** il a la charge de fournir les services du système, de scruter le réseau pour prendre en compte les requêtes de migration d'agents, et d'implémenter le protocole de communication entre les agents de l'hôte ;
- **le réservoir d'événements (*events pool*) :** la communication entre agents du même hôte est réalisée par un mécanisme événementiel ;
- **l'agent :** c'est la brique de base du système ; les services, les fonctionnalités systèmes sont des agents.

[5]Nous avons vu en §3.4 p35 que Java possède un certain nombre de caractéristiques en adéquation avec l'implémentation d'un système de codes mobiles.

[6]D'autres entités de bases sont définies dans AgentOS, mais celles décrites dans cette section nous suffiront pour décrire l'architecture du système.

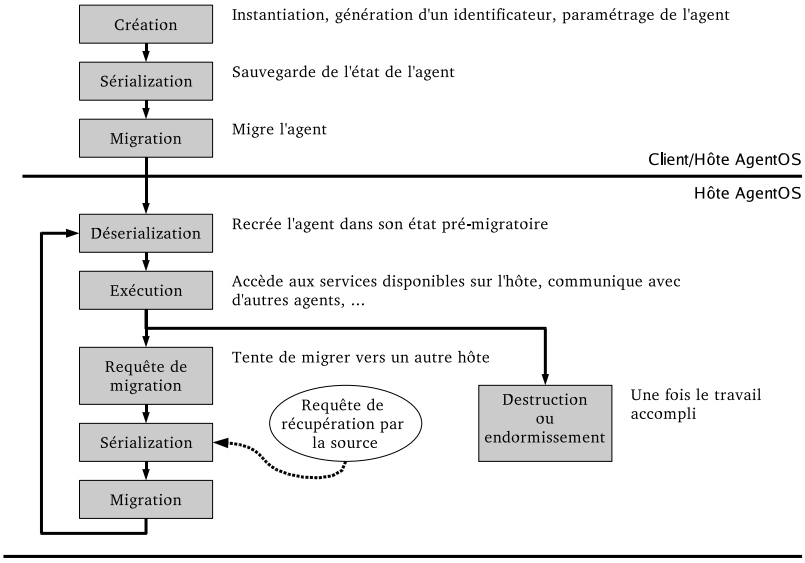


FIG. 4.1 – Cycle de vie d'un agent dans AgentOS.

AgentOS fournit plusieurs fonctionnalités qui sont toujours implémentées à partir de ces entités de bases :

- **services :** plusieurs services sont disponibles dans le système et d'autres peuvent être ajoutés dynamiquement :

  - **liste des agents actifs (*RunningList*) :** ce service permet de découvrir un agent avec lequel on souhaite communiquer ;

  - **historique (*History*) :** ce service permet de mémoriser les agents qui ont migré sur un hôte et donc, utilisé conjointement avec la liste des agents actifs, d'implémenter un algorithme de recherche d'agent ;

  - **répertoire (*Directory*) :** ce service permet d'obtenir la liste des agents actifs sur l'ensemble du réseau AgentOS ; il fournit un algorithme de recherche d'agents ;

  Un service est implémenté à l'aide de l'entité agent, les autres agents qui souhaitent utiliser un service doivent communiquer avec l'agent de service en utilisant le réservoir d'événements ;

- **communication inter-hôtes AgentOS :** cette fonctionnalité est implémentée en utilisant le réservoir d'événements et l'identificateur global d'un agent : un agent contenant le message est envoyé sur l'hôte qui contient l'agent destinataire du message ; l'agent envoyé, une fois à destination, utilise le réservoir local d'événements pour déposer son message.

On le voit, AgentOS utilise au maximum le paradigme *agent mobile*. C'est un système d'agents mobiles qui utilise la migration faible proactive sans gestion automatique de l'espace des données. Le développeur utilise l'interface `IAgent` qui définit les méthodes du listing §4.1 p47 pour gérer l'état de son agent.

```
1 public interface IAgent{
2     ...
3     void onArrival();
4     void onDispatch();
5     ...
6 }
```

Prog. 4.1 – Interface `IAgent` d'un agent dans AgentOS

Aussi étonnant que cela puisse paraître pour un système d'agents conçu à base d'agents, il n'y a pas de mécanisme de sécurité autre que celui fournit par la machine virtuelle : `SecurityManager`. Ce mécanisme n'offre pas de manière standard [7] les moyens de contractualiser les ressources d'un support d'exécution pour assurer la sécurité *intra-support*, ni un mécanisme qui permet d'assurer l'authentification, l'intégrité et la confidentialité des communications *inter-supports*[8] (*cf.* §3.5 p37).

AgentOS ne semble plus être maintenu. La dernière distribution utilisait la version 1.1.3 du JDK (nous en sommes au JDK v1.4.2) et n'était disponible que sous le système d'exploitation Windows (ce qui est étonnant pour une plateforme écrite en Java – probablement pas à 100%).

4.3.2 Aglets

Le terme "aglet" est né de la contraction des deux termes "agent" et "applet". Ce système d'agents [59] créé par Danny B. Lange du laboratoire de recherche IBM de Tokyo est basé sur une API simple qui définit trois abstractions clés :

- **l'aglet :** qui est l'agent mobile proprement dit (classe `Aglet`) ;
- **le proxy :** qui est l'objet utilisé lorsque l'on souhaite manipuler l'aglet associée (classe `AgletProxy`) ;
- **le contexte :** qui est le support d'exécution de l'aglet.

Le proxy sert de *bouclier* : il évite en effet à n'importe quel autre aglet d'avoir une référence directe sur son aglet associée. Ainsi, alors qu'en Java, toutes les méthodes

[7] On peut utiliser l'environnement SAJE [62] à cet effet.

[8] Même si le JDK propose un ensemble de bibliothèques à cet effet.

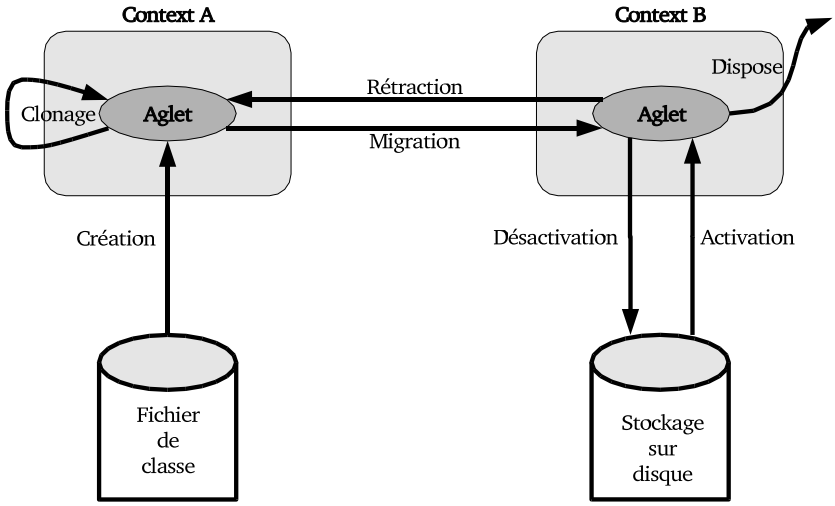


FIG. 4.2 – Le cycle de vie d'une aglet.

publiques sont accessibles, le proxy permet de filtrer les accès aux méthodes de l'aglet. Aussi et surtout, le proxy rend la localisation de l'aglet transparente. Il est capable de transmettre les appels de méthodes à l'aglet associée même lorsque cette dernière ne réside pas dans le même contexte.

Le cycle de vie d'une aglet, illustrée sur la figure FIG. 4.2 p48, est lié à un mécanisme événementiel : le système utilise les *listeners* de l'aglet à chaque événement qui la concerne. Ainsi, une aglet est définie par la classe `Aglet` dont une partie est présentée en PROG. 4.2 p49.

Les listeners d'événements sont enregistrés dans l'aglet à l'aide des méthodes du type `add*Listener()`, et retirés à l'aide de leurs symétriques `remove*Listener()`. La méthode `run()` définit le corps de l'aglet. Cette méthode sera exécutée dans une *thread*[9] à part ce qui rendra l'aglet indépendante. Enfin, la communication entre aglets est assurée par un mécanisme événementiel : l'envoi de messages se fait par l'intermédiaire du proxy (méthode `sendMessage()` pour l'envoi synchrone, `sendAsyncMessage()` pour l'envoi asynchrone) et la réception se fait par la méthode `handleMessage()` de l'aglet automatiquement appelée par le support d'exécution (le *contexte* dans la terminologie des aglets).

La sécurité est assurée par un mécanisme assez complexe d'identification des différents composants du système [107]: les *principals*. Ainsi, on identifie, entre-autres,

[9]Dans notre équipe, nous avons l'habitude de donner le genre féminin au terme anglais *thread*. Il semble que le genre masculin soit aussi utilisé.

```
1  public abstract class Aglet extends java.lang.Object
2                              implements java.io.Serializable {
3  ...
4      // Related to aglet cloning (see clone())
5      public void addCloneListener(CloneListener l);
6      public void removeCloneListener(CloneListener l);
7
8      // Related to aglet migration (see dispatch())
9      public void addMobilityListener(MobilityListener l);
10     public void removeMobilityListener(MobilityListener l);
11
12     public Object clone() throws CloneNotSupportedException;
13
14     public void dispatch(java.net.URL dst)
15            throws IOException,
16                   RequestRefusedException;
17
18     public void dispose();
19
20     public void onCreation(Object object);
21     public void onDisposing();
22
23     public void run();
24
25     public AgletProxy getProxy();
26     public boolean handleMessage(com.ibm.aglet.Message);
27 }
```

PROG. 4.2 – Quelques méthodes de la classe `Aglets`.

- **l'aglet :** c'est l'instance elle-même ;
- **le concepteur de l'aglet :** qui peut être un individu, une organisation, etc. ;
- **le détenteur de l'aglet :** qui est l'entité pour laquelle l'aglet s'exécute.

Chacun de ces *principals* doit définir une *politique de sécurité*. Par exemple, le concepteur de l'aglet définit qu'il a besoin de 10 secondes sur chaque site visité (contexte), mais un contexte peut souhaiter ne donner que 5 secondes pour les aglets provenant d'un hôte particulier. Un ordre total est établi entre les différents *principals*. La spécification des politiques de sécurité est établie en utilisant une grammaire appropriée. Ce mécanisme semble assez compliqué à mettre en œuvre, mais relativement puissant.

Le système Aglets est en open-source, disponible sur Sourceforge [53]. Toutefois, la dernière version disponible date de février 2002.

4.3.3 Grasshopper

La plate-forme d'agents mobiles Grasshopper [158] est développée par GMD FOKUS et IKV++ GmbH. Elle est la première plate-forme – et l'unique à ma connaissance – à être conforme à la spécification de l'OMG : la MASIF [86]. Par ailleurs, Grasshopper a été étendue pour être conforme aux spécifications de la *Foundation for Intelligent Physical Agents* (FIPA), prévu pour les agents intelligents (et donc non-nécessairement mobiles). C'est ici que la frontière entre agents mobiles et agents intelligents devient floue : une même plate-forme pouvant accueillir les deux types d'agents. Notons toutefois que dans Grasshopper, un agent intelligent au sens FIPA ne peut être mobile au sens MASIF : il est nécessairement stationnaire (il doit hériter de la classe `FIPAAgent` sous-classe de la classe `StationaryAgent` qui définit les agents stationnaires).

Grasshopper est largement utilisée dans le domaine des télécommunications au sein du projet européen *Cluster for Intelligent Mobile Agents for Telecommunication Environments* (CLIMATE) [5] qui contient un grand nombre de sous-projets.

L'architecture de Grasshopper est calquée sur celle de la MASIF qui définit une *région* comme un ensemble d'*agences* (*système d'agents* dans la MASIF) elles-mêmes constituées de *places*. Ainsi,

- **une place :** contient un ou plusieurs agents (mobile ou non) ;
- **une agence :** contient une ou plusieurs places ainsi que le noyau du système qui offre les services de communication, de gestion, de persistance, de sécurité et de transport ;
- **une région :** contient plusieurs agences ainsi qu'un annuaire (*registry*) qui permet la recherche et la localisation d'un agent.

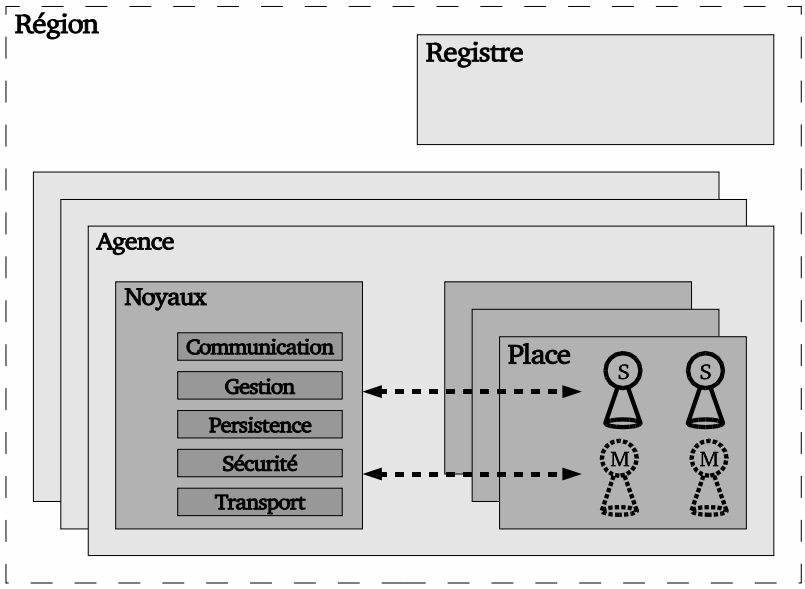


Fig. 4.3 – L'architecture de Grasshopper est calquée sur celle définie par la MASIF.

Cette architecture est illustrée par la figure Fig. 4.3 p51.

Le service de communication est utilisé pour les interactions entre agents, agences et autres entités (comme par exemple le registre). Il est multi-protocole (IIOP, RMI, MAF-IIOP, Socket, RMI/SSL, Socket/SSL), transparent vis à vis de la localisation (c'est à dire que le service de communication s'occupe de trouver le destinataire de l'interaction) et permet des communications synchrones, asynchrones (avec mécanisme de scrutation ou événementiel) et groupées.

En outre, Grasshopper permet de communiquer avec certains services en utilisant les interfaces CORBA certifiées MASIF (`MAFAgentSystem` et `MAFFinder` par exemple).

La communication inter-agents n'est pas couverte par la MASIF. Par contre, la FIPA définit – entre-autres – le langage de communication pour Agents – *Agent Communication Language* (ACL) qui peut donc être utilisé dans Grasshopper.

Enfin, la sécurité externe est assurée par l'utilisation des certificats X.509 et l'encryptage SSL des communications. La sécurité interne repose elle sur l'utilisation d'un SecurityManager Java.

Ainsi, Grasshopper est une plate-forme d'agents mobiles conforme à la MASIF et, dans une certaine mesure, à certains standards de la FIPA. La migration est de type faible et proactive, et sans gestion automatique des données. Cette gestion est réalisée, une fois de plus de manière événementielle, à l'aide de l'interface `MobileAgent` donnée en Prog. 4.3 p52.

```
1 public interface MobileAgent {
2     void beforeMove();
3     void afterMove();
4 }
```

PROG. 4.3 – Interface `MobileAgent` d'un agent dans Grasshopper

Grasshopper est livrée avec une licence commerciale qui autorise seulement son utilisation dans un cadre privé.

4.3.4 Voyager

Voyager [175] est plus un ORB qu'un système d'agents mobiles. Toutefois, sa capacité à rendre tout objet Java sérialisable mobile le place parmi les systèmes d'agents mobiles les plus utilisés.

Voyager utilise la programmation par interface (*design by interfaces*) ce qui lui permet de fournir des implémentations distantes de n'importe quel objet par l'intermédiaire d'un proxy. Par ailleurs, lorsqu'un objet n'implémente pas d'interface, Voyager est capable d'en fournir une à la volée : l'interface contient alors l'ensemble de toutes les méthodes publiques de la classe originale. Enfin, Voyager utilise un mécanisme d'*agrégation dynamique* qui permet d'ajouter des *facettes* à un objet dynamiquement (sans que celui-ci ait été prévu pour). Dans une certaine mesure, Voyager permet par ce biais d'utiliser le paradigme *programmation par aspect* [110]. C'est par ce mécanisme que n'importe quel objet peut devenir mobile en étant rattaché à la facette `IMobility` dont l'interface est donnée en PROG. 4.4 p52.

```
1 public interface IAgent {
2     void moveTo(Object destination);
3     void moveTo(String url);
4 }
```

PROG. 4.4 – La facette `IMobile` dans Voyager qui ajoute l'aspect mobile à tout objet sérialisable.

Par l'intermédiaire de la facette `IAgent` qui étend `IMobility` un objet devient un agent mobile. Cette facette définit notamment la méthode `setAutonomous()` qui permet de rendre l'agent candidat au ramasse-miettes lorsque son paramètre est un booléen valant `false`. Ainsi, la figure FIG. 4.4 p53 illustre le cycle de vie d'un agent

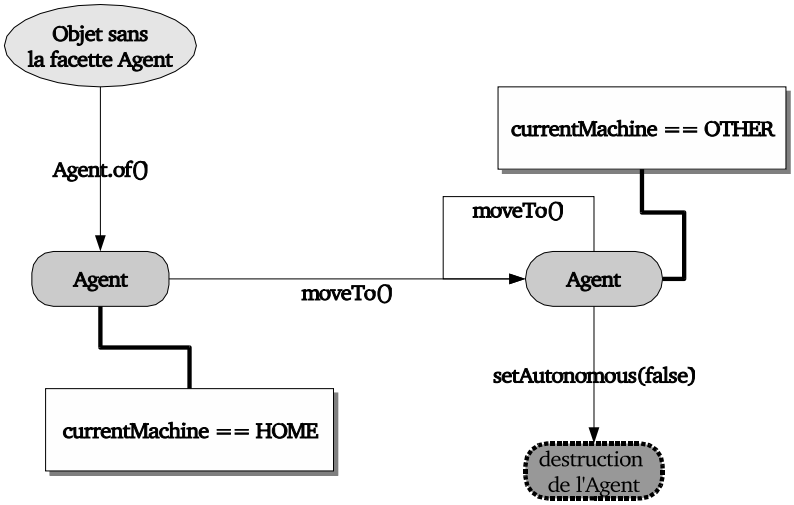


FIG. 4.4 – Un objet devient un agent grâce à la *facette* `IAgent` rajouté par la méthode `Agent.of()`.

dans Voyager.

Voyager fournit des mécanismes de communication synchrone, asynchrone et groupée. Le protocole utilisé par défaut est Voyager Remote Messaging Protocol (VRMP), mais JRMP (RMI), IIOP (CORBA) et ORPC (DCOM) peuvent aussi être utilisés. Enfin, Voyager fournit son propre mécanisme de ramasse-miettes distribué (les objets distants et mobiles dans Voyager ne sont pas nécessairement des objets RMI).

La sécurité est gérée dans Voyager sur trois niveaux différents :

- **le transport :** peut être sécurisé par l'utilisation des sockets SSL ;
- **les objets *forains*:** (c'est à dire dont la classe n'est pas disponible localement) ont des droits limités par l'utilisation d'un `SecurityManager` (adaptation du mécanisme du bac à sable (*sandbox*) des applets Java) ;
- **les utilisateurs :** ont des accès contrôlés par l'utilisation des mécanismes standards Java (paquetage `java.security`).

Voyager est donc un ORB qui supporte le code mobile et en particulier les agents mobiles. La migration est faible, proactive et sans gestion automatique des données. Cette gestion doit donc être effectuée par le développeur à l'aide des méthodes de l'interface `IMobile` donnée en PROG. 4.5 p54.

Voyager est livré avec une licence commerciale. Une version d'évaluation de 30 jours est disponible sur leur site.

```
1  public interface IMobile {
2      void preDeparture(String source,
3                        String destination) throws MobilityException;
4      void postDeparture();
5
6      void preArrival() throws MobilityException;
7      void postArrival();
8  }
```

PROG. 4.5 – La facette `IMobile` dans Voyager qui ajoute l'aspect mobile à tout objet sérialisable.

4.4 Conclusion

Les agents mobiles semblent être une bonne entité d'éxecution. Outre leur mobilité de type généralement faible et proactive, ils offrent un certain nombre de mécanismes qui simplifient le développement des applications (communication ou sécurité par exemple). Nous allons voir que c'est cette entité qui a été retenue pour la plateforme JEM et à ce titre, nous en avons effectué une modélisation en π-calcul. Pour ne pas alourdir d'une partie formelle notre étude des entités mobiles, cette modélisation est donnée en annexe §C.3 p297. Il est néanmoins important de mentionner que c'est cette modélisation qui est à l'orgine du concept de *conteneur actif* que nous présenterons dans la partie §III p75 et qui a aussi fait l'objet d'une modélisation donnée en annexe §D p313.

Chapitre 5

Les processus légers mobiles

Parmi les candidats à la notion d'entité d'exécution mobile dans JEM, la notion de processus légers – que nous appellerons aussi thread dans la suite – semble être un bon choix pour les raisons suivantes :

- une thread représente exactement la notion d'entité d'exécution ;
- une thread est représentée en interne par une structure de données de très petite taille comparée à un processus – généralement, un pointeur d'instruction et une pile – ce qui garantit une migration peu coûteuse (peu de données à transférer pour la restauration).

Nous ne distinguerons pas dans la suite – pour des raisons de simplicité – la notion de threads noyau (processus de poids moyen) et de threads utilisateur (processus de poids léger aussi appelé *green thread*).

5.1 Appel de procédure à distance et migration de processus légers

Dans une certaine mesure, un *appel de procédure à distance – Remote Procedure Call* (RPC) – peut être assimilé à une migration du flot d'exécution d'une machine à une autre. C'est d'ailleurs exactement sous cet aspect que sont présentés les appels de méthodes à distance dans la plateforme JavaParty [96]. Dans un appel de procédure (ou de méthode) à distance, le flot d'exécution est interrompu chez l'appelant – ce qui peut être assimilé à l'arrêt de l'entité d'exécution – tandis qu'un nouveau flot débute sur une machine distante. Ce dernier est la suite logique du flot source et on a bien une migration de flot comme l'illustre la figure Fig. 5.1 p56.

Toutefois, dans un appel de procédure à distance, il n'y a pas véritablement migration de flux mais plutôt *clonage de flux*. En effet le flux d'origine est rarement complètement arrêté : il peut attendre le retour de l'appel distant (cas des fonctions

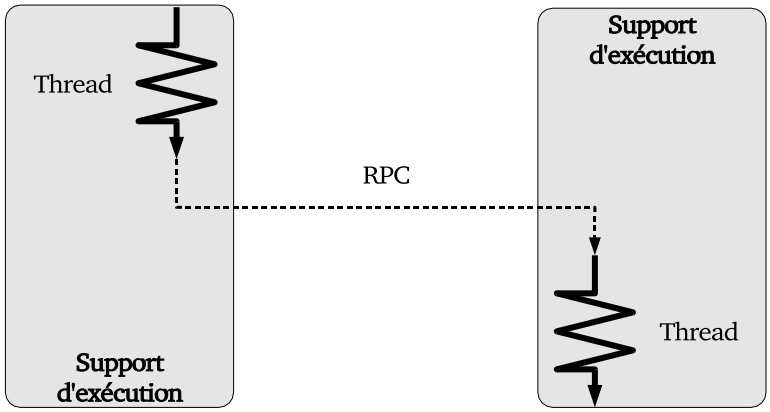


FIG. 5.1 – Un appel de procédure à distance vu comme une migration de flot d'exécution.

par exemple) ou même continuer son exécution avant même que l'appel soit terminé (cas des appels asynchrones). Enfin, il n'y a pas vraiment migration de code dans la mesure où le serveur qui exécute le service demandé par l'appel de procédure à distance contient déjà le code requis pour l'exécution du service (*cf.* les différents paradigmes de conception à la page §3.3 p32).

5.2 Problématique

Migrer une thread en cours d'exécution requiert généralement une migration forte en raison du caractère dynamique, très volatile de cette entité. En effet, à moins de complexifier l'entité thread et de lui rajouter les informations qui permettent :

- d'identifier une thread par une référence ;
- de demander sa migration ;
- de permettre à la thread de réagir au signal de migration ;
- de permettre à la thread de restaurer son état ;

il semble peu raisonnable d'effectuer de la migration faible de threads. Au contraire, la migration forte étant transparente, c'est le système qui gère l'ensemble de l'opération.

Toutefois, nous allons voir que la migration forte de thread pose de nombreux problèmes, du moins en Java.

5.2.1 Récupération du pointeur d'instruction

Migrer une thread requiert la sauvegarde et la restauration de son pointeur d'instruction. Le projet JEM étant écrit en Java[1], il était important de concevoir l'entité d'exécution mobile dans ce langage. Or, la machine virtuelle Java ne permet pas d'accéder à ce pointeur d'instruction. Une nouvelle machine virtuelle avec cette caractéristique fondamentale pour la migration forte de threads était donc à concevoir. A l'époque, le code source de la machine virtuelle Sun n'était pas disponible. Seul Kaffe[7], semblait être adaptable avec de fortes contraintes de compatibilité.

5.2.2 Contexte d'une thread

Le premier et peut être principal problème vient de la gestion du contexte de la thread.

Outre la sauvegarde de l'état d'exécution, nous sommes directement confrontés au problème évoqué en §3.2 p30 concernant la gestion de l'espace des données.

Une thread en cours d'exécution utilise généralement des variables globales, des verrous sur des structures de données partagées avec d'autres threads. Lorsqu'une migration de thread survient, que deviennent les verrous ou les variables de condition qu'elle possédait ? Dans quel état sont les données partagées ? Dans le cadre d'une migration forte, d'autres problèmes sont soulevés par la récupération du contexte. Les verrous et les variables de condition doivent-ils être restaurés ? Dans quel état ? Doivent-ils être partagés ? Avec quelles autres threads ?

Par ailleurs, la migration de threads est souvent justifiée par la faible quantité de données à envoyer : on considère généralement que cette quantité est bornée par la somme de la taille du pointeur d'instruction (4 octets ou 8 octets selon l'architecture) et de la taille de la pile (quelques dizaines de kilos-octets selon le système et/ou les paramètres utilisateurs). Cette taille est très faible comparée à la taille des données à envoyer lors d'une migration de processus (plusieurs méga-octets). Cette caractéristique peut laisser entrevoir des performances intéressantes. Mais c'est oublier toutes les données non-partagées dans le tas qui sont allouées dynamiquement. Pire, l'espace des données devant être restauré après la migration, la quantité de données à transmettre n'est pas seulement bornée par la taille de la pile et des données allouées dans le tas. Il faut aussi transférer toutes les données manipulées par la thread (fichier, structure de données, etc.) partagées au sein du processus avec les autres threads. La taille des données à transférer pour restaurer l'état de la thread après une migration n'est donc pas borné.

[1] L'équipe Système et Objets Distribués fondée par Serge Chaumette a toujours eu une très forte connotation Java, dès les débuts de ce langage.

Par conséquent, la justification précédente n'est pas valide : le coût de la migration d'une thread dépend aussi de la taille de son espace des données et n'est donc pas nécessairement faible. Même en utilisant un mécanisme de référencement à distance pour éviter le transfert, deux problèmes subsistent :

- Le coût de la migration de thread dépend toujours de la taille de son espace des données.
- Nous avons vu que l'utilisation de références distantes n'est pas toujours un bon choix. Il est parfois plus intéressant de copier ou de déplacer une ressource.

5.2.3 Migration réactive de threads

La migration réactive de threads apporte son lot de problèmes que nous allons décrire.

5.2.3.1 Interruption de thread

La migration forte d'une thread comporte plusieurs étapes :

1. suspendre l'exécution de la thread ;
2. sauvegarder son contexte (état d'exécution et espace des données) ;
3. transférer le code et le contexte ;
4. supprimer la thread initiale si le transfert s'est bien déroulé et sinon, plusieurs options sont envisageables :
   - avertir l'utilisateur par un mécanisme événementiel[2] ;
   - restaurer la thread et faire comme si rien ne s'était passé ;
   - avertir l'utilisateur et restaurer la thread.

L'étape de suppression est indispensable, sinon, au lieu de migration, nous aurions *clonage* de threads – qui est aussi une alternative intéressante à la migration dans un cadre d'équilibrage de charge.

La première étape est celle qui est la plus problématique. Il est en effet très délicat d'interrompre une thread en cours d'exécution [66]. En effet, la thread peut être en train de manipuler, une structure de données interne, peut avoir acquis un verrou, peut être en train d'écrire dans un fichier. La suspension de la thread peut placer le processus dans un état instable.

C'est pour cette raison que les méthodes `Thread.stop()`, `Thread.suspend()`, et `Thread.resume()` sont dépréciées depuis le JDK 1.2 [142]. Dans l'API POSIX, le comportement de la fonction `pthread_cancel()` dépend :

[2] Puisque la migration est de type réactive, le programmeur ne peut prévoir à quelles instructions l'erreur va se produire, il faut donc un mécanisme événementiel

- de “l'état d'annulation” (*cancellation state*) de la thread courante qui peut être changé à tout moment en utilisant la fonction `pthread_setcancelstate()`. Cet état peut être :
  - **PTHREAD_CANCEL_ENABLE :** la thread est interruptible ;
  - **PTHREAD_CANCEL_DISABLE :** la thread n'est pas interruptible.
- du “type d'annulation” (*cancellation type*) qui peut être changé à tout moment en utilisant la fonction `pthread_setcanceltype()`. Ce type peut être :
  - **PTHREAD_CANCEL_DEFERRED :** c'est un mode “coopératif”. On demande à une thread de s'arrêter, elle est censée scruter son état d'interruption et y répondre le plus vite possible en effectuant les opérations nécessaires à la restauration d'un état cohérent dans le processus. Normalement, les appels systèmes tel que `read()` et `write()` savent répondre à une telle demande d'interruption proprement.
  - **PTHREAD_CANCEL_ASYNCHRONOUS :** c'est un mode “autoritaire”[3] (*cf.* §C.3.4.2 p309). La thread est arrêtée par le système où qu'elle soit dans son code. Ce mode est très compliqué à utiliser (et déconseillé dans les documentations) car il n'est pas toujours évident de restaurer un état cohérent à partir de n'importe quel point du programme.

Même au niveau du système (API POSIX), il est donc relativement difficile d'interrompre une thread en cours d'exécution.

La migration proactive de thread ne possède pas ces problèmes puisque la thread est l'instigatrice de la migration. Elle est donc à même de décider à quel endroit de son code sa propre migration ne posera pas de problème.

5.2.3.2 Mécanisme de sélection

Migrer une thread de manière réactive implique sa préalable sélection parmi un ensemble de threads. Il faut donc déterminer les critères qui seront utilisés pour cette sélection et les récupérer pour chaque thread. Or, en raison de l'aspect très dynamique de l'entité thread, les informations collectées ont un caractère très éphémère. Ce problème n'est d'ailleurs pas lié à la migration de threads : il est toujours compliqué d'utiliser une information volatile. Cependant, dans le cadre de la mobilité de thread, si la migration de thread est utilisée pour l'équilibrage de charge par exemple, il est absolument indispensable de pouvoir utiliser l'information obtenue après un certain temps. Nous verrons quelles solutions sont mises en œuvre dans certaines plate-formes.

[3] Préemptif ?

Ce problème est encore lié à la migration réactive. Dans une migration proactive, ce problème n'existe pas puisque c'est la thread elle-même qui décide de sa propre migration.

5.3 L'environnement PM^2

L'environnement PM^2 (*Parallel Multithreaded Machine*) [155][4] a été conçu pour faciliter l'implémentation d'applications irrégulières massivement parallèles. L'irrégularité d'une application est définie par la difficulté de connaître, même partiellement, le graphe de précédence de ses tâches. Une application est massivement parallèle si le nombre de ses tâches (utiles) est très important.

5.3.1 Mise en œuvre

PM^2 s'appuie sur deux bibliothèques :

- une bibliothèque de communication efficace : Madeleine ;
- une bibliothèque de threads mobiles en mode utilisateur : Marcel. La bibliothèque de threads Marcel est un sous-ensemble de l'API POSIX auquel les fonctions de migration ont été ajoutées[5]

Le paradigme de programmation utilisé par PM^2 est l'appel de procédure à distance léger (LRPC) : un appel de procédure à distance est exécuté par une thread et non par un processus. Les threads ne sont pas utilisées dans le seul but de recouvrir les communications par le calcul dans une application. Chaque thread représente un processeur virtuel : c'est la *virtualisation des ressources*. Ainsi, les applications sont écrites en utilisant autant de threads que nécessaire. L'efficacité des threads Marcel[6] permet aux applications de gérer l'irrégularité à un second niveau : l'équilibreur de charge. Les applications ne dépendant plus du nombre de processeurs, elles ont donc une meilleure portabilité sur différentes architectures distribuées.

[4]Nous présentons la plateforme que nous avons étudié en 2000, elle a bien entendu changé depuis.

[5]Aujourd'hui, la bibliothèque est une modification sous forme de *patch* de la bibliothèque *LinuxThreads* [125] qui a longtemps été utilisée comme bibliothèque de threads noyau sous linux. Cependant, la *NPTL* [177] tend à la remplacer aujourd'hui.

[6]Les threads Marcel sont implémentées dans l'espace utilisateur ce qui permet d'éviter le passage en mode noyau pour l'ordonnancement par le système et offre ainsi de très bonnes performances en terme de création/destruction et de changement de contexte.

5.3.2 PM^2 dans le contexte du code mobile

PM^2 est un cas particulier puisqu'il ne permet pas à proprement parler de "migration de code". En reprenant la terminologie définie en §3.1 p29, le support d'exécution est un démon qui doit être exécuté en permanence, l'unité d'exécution est la thread Marcel. Toutefois, lors d'une migration de thread, le code n'est pas réellement transféré. Une thread Marcel est liée à une fonction qui est son point de démarrage. Cette fonction est dans le code qui doit être présent sur l'ensemble des machines. PM^2 ne propose pas de chargement de code à la volée. PM^2 est donc un environnement basé sur la migration de *contexte de thread*, et non sur de la migration de code au sens commun du terme. La migration est de type forte et réactive.

5.3.3 Gestion de l'espace des données

Après la migration d'une thread, un réajustement des références de son espace des données doit être effectué, le processus cible n'ayant pas nécessairement la même organisation mémoire que le processus émetteur de la thread (à moins d'utiliser un mécanisme *isoadresse*[7] qui introduit globalement un gaspillage de mémoire très pénalisant sur un grand nombre de nœuds).

PM^2 n'offre pas une gestion automatique de l'espace des données mais un ensemble de fonctions qui permet :

- de placer la thread dans un état migrable/non-migrable (`pm2_disable_migration()`, `pm2_enable_migration()`) ;
- d'effectuer un traitement pre/post migratoire (`pm2_set_{pre,post}_migration_func()`) ;

La gestion de l'espace des données est donc laissée à la charge du développeur. Celui-ci peut, lorsqu'il manipule des données sensibles à la migration (fichier, verrou, etc.) se placer dans un état non-migrable pour éviter d'éventuels problèmes liés au modèle de migration réactif (interruption de thread notamment). Enfin, lorsque la thread migre, elle peut préparer la restauration de son espace de données en implémentant les fonctions pre/post migratoires afin d'utiliser des mécanismes de gestion d'espace de données tels que la copie, le déplacement, ou le référencement distant.

Notons que les pointeurs vers des zones de la pile d'exécution (données privées – qui font partie de l'état d'exécution –) doivent[8] être marqués en utilisant des fonctions particulières (`pm2_register_pointer()`). Afin d'éviter une migration incorrecte, seule une thread dans un état non-migrable peut utiliser ces données marquées. Ce problème des pointeurs dans la migration de threads a été étudié dans [25]. Toutefois, l'environnement PM^2 est prévu pour être utilisé dans le cadre précis des *appels*

[7] Ce que fait la version actuelle.

[8] Ce n'est plus le cas aujourd'hui grâce à l'utilisation du mécanisme d'isoadressage.

de procédures légers (LRPC) qui limite au maximum l'utilisation de pointeurs, et de données partagées.

5.3.4 Gel d'un module

Le problème de sélection de threads dans un modèle de migration réactive (*cf.* §5.2.3.2 p59) est résolu dans PM^2 par le *gel de module*. Un module est l'ensemble des threads d'une machine[9] donnée. Son gel revient à suspendre l'exécution de toutes les threads du module sauf celle qui effectue la demande de gel. Cette dernière peut alors parcourir la liste de threads migrables, sélectionner celle dont la migration est jugée nécessaire, et effectuer la migration.

Si cette solution résout complètement le problème du dynamisme, elle en introduit un autre : sachant qu'une thread peut passer d'un état migrable à non-migrable à volonté[10], il est donc possible que la thread qui parcourt le module à la recherche des threads à migrer (équilibreur de charge par exemple) ne trouve essentiellement que des threads dans un état non-migrable.

5.3.5 Discussion

L'avantage majeur de la migration forte réactive est qu'elle est transparente pour l'utilisateur. Or, nous l'avons vu, PM^2 impose des contraintes soit au système (état migrable/non-migrable, disponibilité du code) soit au développeur (enregistrement des pointeurs, transfert des données allouées dynamiquement). Par conséquent l'entité thread mobile dans PM^2 n'apporte pas la transparence attendue par un système à migration forte réactive.

5.4 Conclusion

Les environnements basés sur des processus légers mobiles sont principalement dédiés au calcul parallèle haute performance. Dans ce cadre, l'entité thread mobile est généralement utilisée comme un outil qui facilite l'écriture de régulateurs de charge. C'est une entité de grain trop fin pour être l'entité d'exécution dans JEM qui n'est pas une plateforme pour le calcul haute performance.

Par ailleurs, on peut se demander si l'utilisation des threads dans un cadre distribué n'est pas contradictoire. En effet, les threads sont souvent utilisées comme un paradigme de communication. Le partage de leur espace mémoire explique leur

[9] En fait, un noeud, une machine pouvant contenir plusieurs noeuds.

[10] Ce qui est d'ailleurs requis lors de la manipulation de pointeurs comme indiqué à la section précédente.

efficacité mais aussi la complexité de leur utilisation. Cependant, dans un système distribué, la mémoire n'est justement pas partagée. Aussi, pourquoi utiliser le processus léger comme l'entité de base d'un paradigme de programmation d'applications distribuées ? En réalité, les processus légers sont de plus en plus utilisés dans les applications distribuées pour favoriser le recouvrement des communications par du calcul. Mais nous avons vu que dans le cadre fixé par PM^2, ils apportent une virtualisation des ressources (nombre de processeurs virtuels quasi-illimités) ce qui favorise la portabilité des applications (ne dépendent pas d'une architecture matérielle donnée). Il est donc important de se conformer au paradigme de programmation de PM^2, l'appel de procédure à distance léger, pour bénéficier de ces avantages. En particulier, dans un cadre local où les threads PM^2 sont utilisées dans un paradigme de communication intra-processus, de nombreuses contraintes limitent fortement leur usage (problèmes liés aux threads utilisateurs : appels bloquants sans support du noyau[11], et support des architectures SMP essentiellement.).

[11]Un patch du noyau linux est nécessaire pour bénéficier du modèle des *Scheduler Activations* [18].

Chapitre 6

Les cartes à microprocesseur

Notre tour d'horizon du domaine du code mobile ne serait pas complet sans étudier le domaine des cartes à microprocesseur. En effet, la carte à microprocesseur est de plus en plus considérée dans le domaine du code mobile et plus encore dans le domaine des systèmes et objets distribués. Par ailleurs, depuis l'arrivée de Damien Sauveron dont les travaux de thèse s'articulent autour de la sécurité de la JavaCard[143], notre équipe développe une expertise reconnue dans le domaine de la carte à microprocesseur [49, 184, 181].

Nous allons voir qu'il existe trois types de carte à microprocesseur : les cartes base de données, les cartes mono-applicatives et les cartes multi-applicatives. Nous illustrerons par des exemples que la carte à microprocesseur est utilisée dans un cadre mobile. Nous montrerons ensuite les analogies et les différences fondamentales entre le concept de mobilité "classique" et le concept de mobilité dans le cadre des cartes à microprocesseur.

6.1 Les cartes base de données

La vocation première de la carte à microprocesseur est de fournir un support de données portable. Les premières générations de carte contenaient essentiellement un système de fichier ou une base de données [68] qui permettait l'interrogation, l'ajout et la suppression de données à partir d'un terminal. Dans ce cadre, la carte joue un rôle passif et ne permet pas d'effectuer des tâches complexes. Malgré tout, certains travaux ont montré que ces cartes pouvaient être utilisées dans un cadre de système à code mobile.

6.1.1 Chiffrement dans le système de fichier CFS

Le système de fichier CFS [32](Cryptographic File System) permet de chiffrer le contenu d'une arborescence (fichiers et noms de fichiers) de manière transparente pour l'utilisateur. Ce dernier utilise un mot de passe, et par l'intermédiaire de la commande `cattach` demande le chiffrement d'une arborescence donnée, par exemple `/usr/alice/secrets`, et son accès par un chemin donné, par exemple `/crypt/sec`. La gestion des mots de passe est toujours un problème dans le domaine de la sécurité : le choix d'un bon mot de passe (donc difficile à mémoriser) contraint souvent l'utilisateur à utiliser le même mot de passe pour l'ensemble de ses services (compte, web, mail, CFS, etc) ou à le noter quelque part.

Aussi, la nature sécurisée de la carte à microprocesseur peut servir à l'identification et éviter dans une certaine mesure cette gestion des mots de passe.

C'est exactement l'extension que propose SC-CFS [105]. La carte est utilisée pour générer des clés à partir d'une clé maître (*master key*) générée aléatoirement et stockée dans la carte. Chaque fichier est chiffré par la machine cliente avec une clé différente ce qui assure un maximum de sécurité. L'utilisateur n'a qu'un seul mot de passe à retenir : le code PIN de sa carte. La machine cliente peut d'ailleurs être un simple *proxy* vers un serveur NFS. Les données sont chiffrées chez le client, le serveur ne stocke que les données chiffrées. Dans ce cadre, l'aspect mobile est important : les données d'un utilisateur ne peuvent être déchiffrées que sur un poste client capable de lire sa carte. Malheureusement, les mauvaises performances du système SC-CFS, liées à la performance de la carte pour la génération des clés et les appels d'entrés/sorties ne semblent pas permettre une utilisation "grandeur nature".

6.1.2 La carte à mémoire étendue

Dans *the vault* [31], la carte représente un individu et doit permettre lors d'une consultation médicale chez un médecin quelconque (d'où l'aspect mobile) l'identification du porteur et l'utilisation de ses informations médicales de manière sécurisée. Ces informations ne peuvent pas être stockées dans la carte en raison de leur taille (images radios, ou scanners par exemple). Le stockage est réalisé sur un serveur Web particulier[1]. La carte contient seulement des références (URLs) afin d'utiliser le moins de mémoire possible.

L'utilisation du protocole RKEP [33] autorise le chiffrement et le déchiffrement efficace sur la machine hôte de la carte (le médecin par exemple), le stockage étant effectué sur le serveur de manière chiffrée.

[1]Notons que le stockage sur la machine hôte n'aurait pas de sens !

6.1.2.1 Le protocole RKEP

Le protocole RKEP permet à une carte à microprocesseur, sécurisée mais de faible bande passante, de fonctionner comme une unité de chiffrement de forte bande passante pour une machine non-sécurisée mais avec un processeur rapide.

La carte contient une clé k secrète. L'idée est de demander à la carte de fournir une clé K_P qui est fonction de sa clé interne et du bloc de bits P à chiffrer. Cette clé K_P est calculée à l'aide de l'algorithme de chiffrement contenu dans la carte (DES [161] en l'occurrence, mais pas nécessairement). C'est cette clé qui sera utilisée par la machine hôte pour le chiffrement réel du bloc P – la clé k n'étant jamais délivrée à l'extérieur de la carte. Ainsi, la carte joue un rôle semi-actif:

- elle contient la clé secrète k;
- elle contient l'algorithme de chiffrement qui permet de délivrer K_P;

Les données cryptées par l'algorithme RKEP en utilisant une carte donnée, ne peuvent être décryptées qu'en utilisant la même carte. Les performances obtenues par cet algorithme ont permis de remplacer le mot de passe dans le système de fichier CFS par la carte à microprocesseur.

La carte à mémoire étendue, en utilisant ce protocole exploite la mobilité géographique de la carte puisque l'utilisation des informations médicales stockées à distance sur le serveur web ne peut avoir lieu que sur la machine hôte de la carte.

Toutefois, l'analogie avec le code mobile présenté précédemment n'est possible qu'en considérant les nouvelles générations de cartes mono et multi-applicatives.

6.2 Les cartes mono et multi-applicatives

Les cartes mono-applicatives contiennent un code encarté figé (stocké dans une mémoire de type ROM), les données étant généralement structurées en fichiers. Le système d'exploitation et l'application sont étroitement liés, formant un *masque*. Les récentes générations de cartes telles que JavaCard [143] ou SmartCard pour Windows [56] sont dites *multi-applicatives*. Trois concepts fondamentaux définissent ce type de carte:

1. la *coexistence* de plusieurs services,
2. l'*évolution dynamique* du contenu applicatif de la carte,
3. la *coopération* entre les services.

Le premier concept évite la prolifération de cartes. Le second permet à la carte d'être vue comme une *structure d'accueil* pour code mobile. Le dernier permet à des services différents de partager des informations (points cumulés par le porteur par exemple). Nous allons voir quelques travaux qui utilisent la carte à microprocesseur dans un cadre mobile.

6.2.1 Gestion sécurisée de la mobilité de l'utilisateur

Ce type de carte est utilisé dans la plate-forme GUM-E[2] [92] pour sécuriser l'accès à un terminal générique anonyme (TGA) tel qu'une borne internet dans un lieu public par exemple. Un utilisateur nomade souhaite accéder à un service auquel il a souscrit auprès d'un prestataire de service. Ce service, privé et personnalisé, est accessible à travers le réseau Internet via le World Wide Web. L'utilisateur n'emporte pas avec lui une machine de connexion au service. L'utilisateur devra donc au fil de ses déplacements trouver des points d'accès à Internet pour accéder à son service.

Dans ce cadre, le terminal utilisé n'est pas "de confiance": un espion peut par exemple avoir été inséré dans le terminal volontairement par le propriétaire du terminal ou involontairement par le biais d'un logiciel contenant des *spywares*; les entrées/sorties sur le terminal peuvent donc être surveillées voire détournées. Par exemple, le terminal peut modifier le montant d'une transaction bancaire saisie par l'utilisateur tout en simulant un affichage correct. Pour éviter ce problème, l'approche GUM-E[2] distingue les données *sensibles* des données *communes*. Les données sensibles sont celles qui sont confidentielles comme des soldes de comptes par exemple. Les données communes sont des données non confidentielles utilisées pour la présentation des données sensibles. Par exemple, dans la sortie suivante :

```
Votre solde est de : 3141 euros
```

les chaînes de caractères "`Votre solde est de :`" et "`euros`" sont des données communes. Par contre, la chaîne "3141" est une donnée sensible.

L'approche est de déporter l'ensemble du *contexte d'utilisation* – qui représente un utilisateur vis à vis du système – dans une carte à microprocesseur.

L'ensemble du traitement et de la formulation des requêtes est effectué par une application encartée dédiée à l'accès aux services. Elle chiffre les messages et les envoie au TGA. Cet envoi contient en outre le destinataire du message ce qui permet au TGA de jouer le rôle de routeur. Les serveurs destinataires traitent les messages et retournent au TGA une réponse en suivant le même protocole.

Le TGA n'a donc pas la possibilité d'exploiter les données transmises entre la carte et le serveur. Enfin, les entrées/sorties sont traitées de façon particulière : les données sensibles passent par un canal fiable et sécurisé tel qu'un *CardComputer*[2]. Les sorties sur le TGA sont donc transformées pour ne faire apparaître que les données communes. Un mécanisme particulier doit permettre à l'utilisateur de repérer dans le dispositif d'affichage du TGA les données sensibles qui sont présentées sur le dispositif externe sécurisé. Par exemple un étiquetage de champs comme dans l'exemple suivant :

[2]Lecteur de carte intégrant un petit afficheur et un clavier associé à un dispositif de sécurité.

sur le TGA : `Votre solde est de : #1 euros`
sur le dispositif sécurisé: `#1: 3141`

Dans cet exemple, c'est la mobilité du code encarté qui est exploitée, pas seulement la mobilité des données : chaque service est associé à une application particulière.

6.3 Situation de la carte à microprocesseur

La carte à microprocesseur joue un rôle particulier dans le domaine de la mobilité de code [91]. En effet plusieurs spécificités sont à considérer :

- la mobilité *géographique* de la carte ;
- *l'hébergement* de code dans les nouvelles cartes dites *multi-applicatives* ;
- la sécurité intrinsèque de la carte.

6.3.1 Mobilité géographique

La mobilité géographique ne peut être comparée à la mobilité de code classique qui a été exposée précédemment. En effet, dans cette dernière, le code migre entre des nœuds reliés par un réseau (que ce soit directement ou via d'autres nœuds). Par contre, dans le cadre des cartes à microprocesseur, le code "saute" de nœud en nœud par *sauts discrets* [91], sans suivre de lien physique.

6.3.2 Mobilité du code

Dans les cartes multi-applicatives, la carte peut être considérée comme un *support d'exécution* : elle est capable d'accueillir un code et de l'exécuter. Toutefois, à l'heure actuelle, la carte est considérée comme un serveur : elle répond à des requêtes APDU (*Application Programming Data Units*) et ne peut directement envoyer des requêtes à l'extérieur ce qui en fait un composant passif.

Notre équipe propose de rendre la carte plus active en la considérant cliente [49]. Ainsi, elle peut directement participer à l'activité d'un système réparti et peut donc être considérée comme une machine à part entière.

6.3.3 Sécurité *intra-support d'exécution*

La sécurité est une caractéristique fondamentale de la carte à microprocesseur qui fonde son existence propre. Nous avons vu en §3.5 p37 que la mobilité de code impose des mécanismes de sécurité pour protéger à la fois le code migrant du support d'exécution, le support d'exécution du code migrant et les codes migrés entre-eux.

Si la carte garantit que les applications encartées ne sont pas accessibles depuis l'extérieur (sécurité intrinsèque), il est nettement plus délicat de garantir l'innocuité des applications encartées. Deux grandes catégories de mécanismes sont utilisées dans ce but :

- **sécurité par adressage :** l'accès aux zones de mémoire en lecture ou en écriture est protégé. La protection peut être matérielle grâce à des unités de gestion de la mémoire (*Memory Management Unit* – MMU) ou logicielle par un dispositif d'isolation [219].
- **sécurité par typage :** les opérations sur les données sont définies en fonction de leur type (opération arithmétique sur les entiers, référencement sur les tableaux, etc.). Dans ce cas, le programme ne doit pas pouvoir accéder à la mémoire directement. C'est l'approche utilisée dans Java.

Chapitre 7

Conclusion

La mobilité et la migration sont deux termes qui ont été très utilisés au début de ma thèse. L'idée que des programmes pouvaient se déplacer de machine en machine a été le moteur de beaucoup de travaux dans ce domaine. Le domaine semblait très prometteur mais force est de constater qu'il n'y a pas aujourd'hui sur l'ensemble du réseau Internet des programmes dont la mobilité est la caractéristique principale[1]. Il serait donc tentant de penser que la mobilité impose trop de contraintes pour être employée à grande échelle. Ces contraintes ne sont pas nécessairement d'ordre technique comme nous l'avons vu en §4.2.2 p43. Cependant nous pensons qu'au niveau du développement, il manque une abstraction qui permet d'exprimer de manière simple et canonique le concept de mobilité de code. Il nous semble que le concept de *conteneurs actifs* est une telle abstraction.

[1] Sauf peut-être les vers ?

Troisième partie

Les conteneurs actifs

Chapitre 8

Présentation des conteneurs actifs

La partie précédente s'est attachée à donner un tour d'horizon du domaine du code mobile. Nous avons vu que dans ce domaine, les agents mobiles tiennent une place à part. Nous avons étudié différents systèmes d'agents mobiles et fourni une modélisation en π-calcul (*cf.* annexe §C.3 p297). Cette modélisation fait apparaître une nouvelle structure de données que nous appellerons *conteneur actif* et que nous présentons ici.

8.1 Introduction

Dans les systèmes d'agents mobiles, les agents peuvent passer d'un serveur à un autre au gré de leur migration. Un serveur d'agent *contient* un agent. Cette relation de contenance est d'une importance fondamentale. En effet, les autres agents du réseau n'ont jamais de référence directe sur l'agent. Ils doivent nécessairement utiliser le serveur d'agents ou les services qu'il héberge pour pouvoir communiquer avec un agent. Lorsque deux agents communiquent, ils délèguent au serveur d'agents le soin d'effectuer la communication (réservoir d'évènements dans AgentOS, ORB dans Voyager, service de messagerie dans les Aglets, service communication dans Grasshopper).

Évidemment, pour pouvoir communiquer entre eux, les agents doivent pouvoir s'identifier. Un identifiant d'agent doit donc permettre :

- de trouver le serveur qui contient l'agent ;
- d'identifier l'agent sur ce serveur.

On voit donc apparaître une table de correspondance dans chaque serveur d'agents qui lui permet de retrouver un agent donné. Cette table doit permettre l'association entre un agent et une clé. Ainsi, si cette association est bijective, un agent est identifié de manière unique en connaissant :

- l'identifiant du serveur ;

- la clé de l'agent.

8.2 Définition du conteneur

Du point de vue d'un système d'agents, la notion de migration d'un agent peut être remplacée par deux opérations élémentaires :

- restreindre la vue de son serveur local de telle manière que l'agent partant n'y apparaisse plus ;
- augmenter la vue du serveur destinataire de la migration de telle sorte que l'agent entrant y apparaisse.

Nous pouvons donc utiliser la notion de conteneur pour simuler celle de vue d'un serveur d'agents. Définissons donc un serveur d'agents comme un conteneur ayant pour interface :

```
void put(Object key, Agent agent);
void remove(Object key);
Agent get(Object key);
```

La méthode `put()` permet d'insérer un agent dans un serveur d'agents en l'associant à une clé donnée. La méthode `remove()` permet de supprimer un agent du serveur qui le contient. Enfin, la méthode `get()` permet de récupérer une copie de l'agent.

8.2.1 Migration d'agents

Une migration d'agent s'écrit donc très simplement avec une API de conteneur. Considérons deux serveurs `s1` et `s2`, les instructions suivantes expriment une migration d'agent du serveur `s1` vers `s2` :

```
Agent a = s1.get(key);
s1.remove(key);
s2.put(key, a);
```

Le lecteur attentif aura remarqué qu'il y a une migration supplémentaire dans le code précédent. En effet, la première ligne rapatrie le code localement (chez le client), puis le retire de `s1` et enfin l'insère dans `s2` comme l'illustre le schéma §8.1 p77. En réalité, on peut considérer que ce problème n'a pas lieu d'être puisque c'est l'agent qui décide de migrer dans un cadre de migration *proactive*[1]. Aussi, la migration précédente s'écrit :

[1] Toutefois nous proposerons une solution élégante à ce problème en §20.1.3 p236 qui fera intervenir la communication inter-agents.

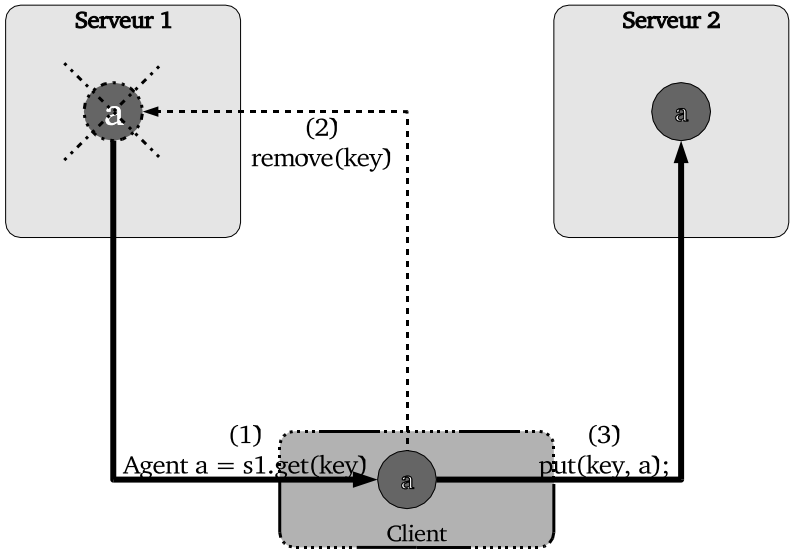


Fig. 8.1 – Une migration de code supplémentaire est induite par l'expression `get()`.

```
s1.remove(myKey);
s2.put(key, this);
```

Donc, notre interface de conteneurs nous permet d'exprimer la migration. Cependant, nous avons encore deux problèmes à régler : la communication inter-agents et la gestion de l'espace des données. Ces problèmes vont trouver une solution avec la caractéristique singulière de nos conteneurs : leur comportement *actif*.

8.2.2 Comportement *actif* du conteneur

Dans le modèle agents mobiles, nous l'avons vu, la communication inter-agents passe toujours par le biais du serveur d'agents. Par conséquent, notre conteneur doit fournir un mécanisme qui permet la communication entre agents résidants dans des conteneurs différents. Nous allons nous baser sur le mécanisme bien connu des appels de méthodes[2] en rendant *active* notre structure de données "conteneur". Un conteneur actif est un conteneur – au sens défini précédemment – qui possède en outre la faculté d'invoquer des méthodes sur les objets qu'il contient par l'intermédiaire de l'interface suivante :

```
void call(Object key, Method m, Object[] args, Result r);
```

[2]Notons que l'abstraction naturelle de l'appel de méthode est le passage de message ; il s'étend très naturellement au cadre distribué. Cela justifiera que nos conteneurs actifs seront accessibles à distance.

Cette méthode invoque la méthode `m` spécifiée de l'objet associé à la clé `key` dans ce conteneur avec les arguments `args`, et retourne le résultat dans un objet `r`. C'est cette méthode `call()` qui rend notre conteneur *actif*. Contrairement aux structures de données habituelles (passives), cette dernière offre une fonctionnalité supplémentaire : la possibilité de communiquer avec les objets stockés par l'intermédiaire du conteneur. C'est ce qui nous permettra de supporter la communication inter-agents et de fournir un mécanisme événementiel pour la gestion de l'espace des données.

En effet, pour qu'un agent a_1 communique avec un agent a_2, il suffit qu'il connaisse le conteneur c qui contient a_2, la clé associée à a_2 dans c, et qu'il invoque la méthode de l'agent via le conteneur.

De même, dans notre système d'agents à base de conteneurs actifs, le mécanisme événementiel pour la gestion de l'espace des données consiste à invoquer : avant chaque migration, la méthode `onMigrating()` sur le conteneur source, et après la migration, la méthode `onMigration()` sur le conteneur destination.

Enfin, la méthode `m` spécifiée dans l'appel à `call()` est invoquée de manière *asynchrone*. Cette particularité rajoute de la concurrence à notre système – condition *sine qua non* à l'obtention d'un système d'agents mobiles autonomes et indépendants. Nous verrons par ailleurs que cette caractéristique a une influence considérable dans ces travaux puisque la partie §IV p123 est entièrement consacrée à la gestion de la concurrence.

8.3 Les conteneurs actifs et les agents mobiles

Nous voyons donc que le concept de conteneur actif est en réalité le concept sous-jacent utilisé implicitement dans les systèmes d'agents mobiles. Toutefois, exprimer un système d'agents mobiles à l'aide du concept de conteneur actif permet une décomposition en couches abstraites plus fines. Cela facilite généralement la modélisation : on réalise un modèle du concept de conteneur actif, devenant ainsi une "boîte noire" au dessus de laquelle un système d'agents mobiles peut être modélisé. Une telle entreprise a été effectuée ; elle est donnée en §D p313.

Chapitre 9

Appel de méthodes à distance

Le domaine d'applications de notre entité *conteneur actif* n'est pas limité à celui des agents mobiles. En effet, si ce concept est en fait le paradigme sous-jacent au modèle agents mobiles, il peut être étendu directement au domaine plus vaste des codes mobiles, et même des objets distribués. Dans ce cadre, nous allons dresser un état de l'art.

9.1 Standards

Dans le domaine des applications orientées objets distribuées, un certain nombre de standards existent et sont très utilisés. Les trois principaux sont RMI [191], CORBA [217, 162], et COM/DCOM[1]. Ils utilisent le modèle des *objets fragmentés* [133] dans lequel un objet distant est composé de quatre parties distinctes :

- **le fragment métier :** c'est lui qui contient le *code fonctionnel* de l'objet distant ;
- **le proxy :** il est capable d'effectuer du calcul localement ou de transmettre les appels de méthodes à sa référence distante correspondante en les encodant dans des paquets réseaux (un *stub* est un cas particulier de proxy qui n'effectue aucun calcul localement et qui est donc réduit à la fonction de communication) ;
- **le squelette :** il décode les appels qui arrivent du réseau et les traduit en appels de méthodes ;
- **le fragment serveur :** il écoute sur le réseau les appels de méthodes entrants, utilise le squelette pour le décodage et invoque la méthode spécifiée.

Lorsqu'il n'y aura pas d'ambiguité, nous considérerons le fragment serveur et le squelette comme un tout que nous appellerons simplement *la partie serveur*.

[1] Que nous connaissons moins bien et qui ne sera donc plus présenté dans la suite.

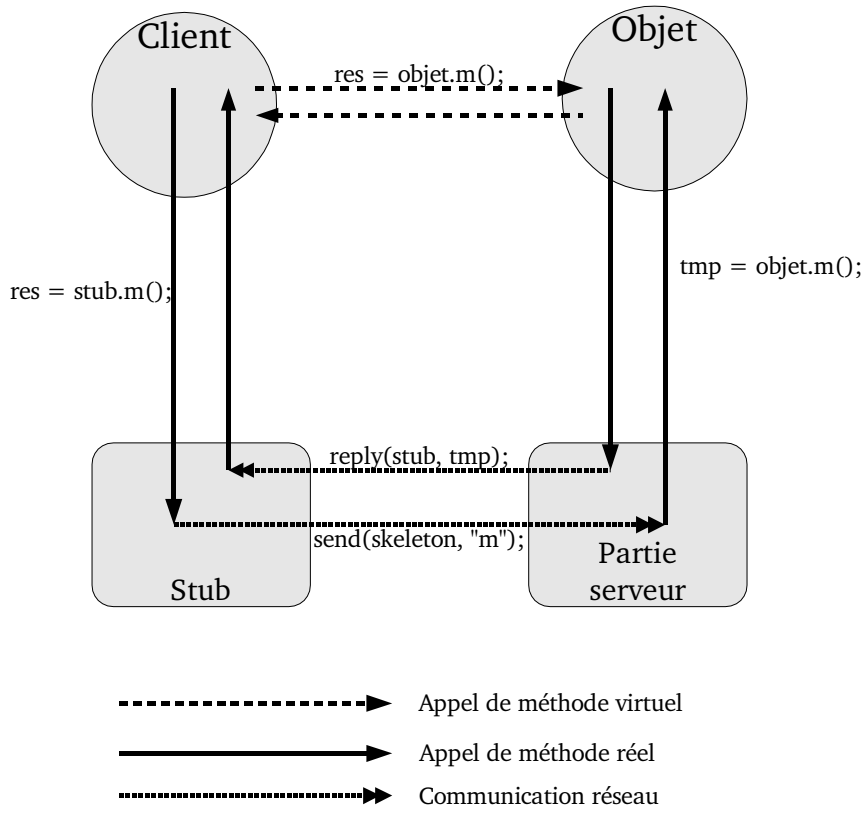


FIG. 9.1 – Java RMI utilise le paradigme Stub/Squelette.

En général, une interface doit être déclarée (IDL en CORBA, `java.rmi.Remote` en RMI) et implémentée par l'objet distant. Les fragments proxy et squelette sont générés par un compilateur (IDL compiler et `rmic`). Par exemple, le compilateur `rmic` génère à la fois le fragment squelette[2] et le proxy – un *stub* en fait. Ce *stub* implémente l'interface du fragment métier qui généralement hérite de la classe `java.rmi.server.UnicastRemoteObject`. Chaque appel de méthode du *stub* est donc transmis à la partie serveur qui invoque finalement la méthode correspondante comme l'illustre la figure FIG. 9.1 p80.

Mis à part le fragment métier, les trois autres parties sont :

- héritées[3] par la classe métier – comme le fragment serveur (`UnicastRemoteObject`) des objets RMI[4] ;
- générées par un compilateur – comme le stub et le squelette RMI créés par le compilateur `rmic`.

[2]Depuis le JDK v1.2, l'utilisation de la réflexion dans les appels RMI élimine le fragment squelette.

[3]La composition peut aussi être utilisée mais le fragment métier doit "récupérer du code" par un moyen ou un autre.

[4]Si cet héritage n'est pas nécessaire, il est conseillé dans la documentation.

Ces solutions sont statiques. Elles demandent au développeur de coder la partie métier en fonction de l'aspect *distant* de son objet. Il doit donc écrire un certain nombre de lignes de code non-fonctionnelles afin de rendre un objet distant. Il est encore plus complexe de rendre distant un objet non-RMI – surtout lorsque l'on ne possède pas les sources de la classe associée aux objets que l'on souhaite rendre distants.

Par ailleurs, la migration de code est un problème qui n'est intégré dans aucun de ces standards[5] : si les objets peuvent migrer en étant passés en paramètres à une méthode distante (pour RMI), leur migration n'est pas directement visible de l'extérieur. Par ailleurs, il n'est pas possible de migrer un objet distant. Lorsqu'un tel objet est passé en paramètre à une méthode, c'est son *stub* qui est envoyé.

Aussi, plusieurs travaux ont tenté d'améliorer les choses.

9.2 CentiJ

Afin de faciliter la réutilisation de classes dans un cadre distant, le projet CentiJ [129] génère une architecture de classes basée sur le *bridge design pattern* [78]. Il permet à n'importe quel objet de devenir distant en utilisant RMI. S'appuyant sur le mécanisme de la réflexion, CentiJ n'a pas besoin du code source des classes que l'on souhaite rendre distantes. Cette solution apporte ainsi le dynamisme que nous recherchons.

Toutefois, CentiJ se limite à l'utilisation du protocole RMI. Ainsi, il n'est pas possible d'utiliser une autre API (CORBA par exemple) ou une autre couche de transport (UDP, ou Myrinet/BIP par exemple).

Par ailleurs, CentiJ ne gère pas les exceptions réseaux : il considère que le réseau est fiable. Si une exception de type `java.rmi.RemoteException` est levée, le client n'en aura absolument pas connaissance. De ce point de vue, on peut considérer que le code généré rend transparent la gestion des communications réseaux, mais cette transparence est *faible* : en ne prenant pas en compte les exceptions réseaux, on ne gère pas l'une des caractéristiques essentielles des systèmes distribués : *les pannes partielles* (*cf.* §10.2.3 p100).

De plus, l'aspect mobile n'apparait pas dans CentiJ. La mobilité du code n'est pas exploitée au même titre que RMI. Enfin, les communications sont nécessairement synchrones.

[5]La sérialisation est un mécanisme plus général que la migration d'objet. La migration utilise la sérialisation. Un objet migré reste potentiellement accessible du monde extérieur, ce qui n'est pas nécessairement le cas d'un objet desérialisé.

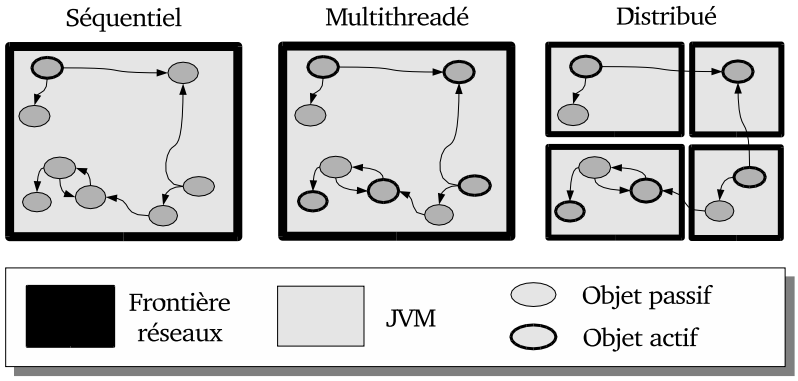


FIG. 9.2 – Organisation structurelle d'une application ProActive.

9.3 ProActive

ProActive [41] est un outil de distribution d'applications écrit entièrement en Java utilisant le modèle *objet actif* que nous décrirons en §12.4.1.1 p134.

Une application ProActive est structurée en sous-systèmes. Il y a un objet actif (et donc une thread) par sous-système et un sous-système par objet actif (ou thread). Chaque sous-système est donc composé d'un objet actif et d'un certain nombre d'objets passifs (y compris zéro). La thread d'un sous-système exécute seulement les méthodes de ses objets. Il n'y a pas d'objet passif partagé entre sous-systèmes. ProActive est prévu pour être utilisé dans un cadre local ou dans un cadre distribué. La figure FIG. 9.2 p82 illustre les différentes topologies possibles et leurs implications sur l'application.

Généralement, un objet actif est distribué en désignant à sa création sa localité[6]. L'aspect distribué n'intervient plus une fois les objets rendus actifs. Cette approche permet de développer des applications distribuées tout en les programmant de manière séquentielle.

Cette transparence est rendue possible par l'utilisation intensive des mécanismes d'héritage, de redéfinition de méthodes et de la *réflexion*. En effet, l'objet *futur* [222] retourné par un appel de méthode sur un objet actif est instancié à partir d'une sous-classe de la classe retournée par la méthode appelée. Cette sous-classe implémente le mécanisme d'*attente par nécessité*. Ce mécanisme permet d'écrire le fragment de code §9.1 p83.

ProActive a cependant quelques limitations :

1. la création de classes à la volée pour l'héritage d'objets rend l'exécution assez lourde, du moins dans la phase d'initialisation des objets actifs ;
2. la classe doit nécessairement avoir un constructeur sans argument ce qui limite

[6]La localité d'un objet actif est généralement équivalente à une machine.

```
1  // Créé ou récupère un objet actif
2  // L'instance retournée est une sous-classe de la
3  // classe d'origine. Cette sous-classe implémente
4  // le proxy
5  MyObject o = (MyObject) ProActive...
6
7  // Invoque une méthode de manière asynchrone
8  // et potentiellement distante (dépend de la localisation
9  // de l'objet actif)
10 MyResult r = o.m();
11
12 doSomethingElse(); // Durant l'exécution de m()...
13
14 // 'r' est en fait une instance d'une sous-classe de
15 // 'MyResult' et implémente le mécanisme
16 // d'attente par nécessité.
17 r.foo(); // appel bloquant si 'm()' n'est pas terminé
```

Prog. 9.1 – Appel asynchrone (et potentiellement distant) transparent dans ProActive

le nombre de classes utilisables : sur les 3 950 du JDK v1.4[7], seules 57% sont publiques dont 42% sont dans ce cas ;

3. les classes déclarées `final` (9% des classes publiques) ne peuvent pas être dérivées, ce qui empêche leurs instances de devenir actives ou la création d'objets *futurs* de ce type ;
4. les méthodes déclarées `final`[8] (5% des méthodes publiques des classes publiques) ne peuvent être redéfinies, ce qui empêche les objets futurs de gérer le résultat des appels asynchrones ;
5. les méthodes qui retournent un type primitif (27% des méthodes publiques des classes publiques) ne peuvent être surchargées pour retourner un objet *futur*, elles sont donc nécessairement synchrones ;
6. les objets qui accèdent directement aux champs publics des objets futurs ne peuvent être bloqués en attente du résultat ;
7. le problème habituel lié à la sémantique du mot-clé `static` dans un cadre distribué n'est pas résolu : si un champ déclaré `static` est modifié, la nouvelle

[7]Seules les classes dont le nom contient le préfixe `java` ont été considérées.

[8]Il est fréquent en Java de déclarer une méthode `final`. Si ce mot clé doit être utilisé avec parcimonie lorsqu'une méthode ne doit pas être redéfinie, il est souvent employé pour des (mauvaises) raisons d'efficacité.

valeur devrait être répercutée sur toute les machines virtuelles qui ont chargé la classe ;

8. les objets qui accèdent directement aux champs `public` d'un objet actif sont problématiques : si l'objet actif est distant, le champ auquel on accède effectivement est celui du proxy et non celui de l'objet actif, d'où un problème de cohérence ;
9. les objets passifs partagés par des objets actifs (dans le cas séquentiel) sont copiés dans chaque nœud lors de la distribution. Ils ne sont donc plus partagés ;
10. un objet actif est traversé par une et une seule thread, ce qui limite le degré d'asynchronisme : si plusieurs appels sont effectués par des clients différents, ils seront exécutés de manière séquentielle ;
11. les méthodes déclarées avec `throws` (20% des méthodes publiques des classes publiques) sont implicitement synchrones même si les autres méthodes restent implicitement asynchrones[9] ;
12. la récupération d'une exception non contrôlée (et non déclarée) après un appel asynchrone nécessite une synchronisation explicite ;

Les problèmes n°1 p82 à n°6 p83 sont liés à l'utilisation de l'héritage pour l'implémentation du mécanisme de *transparence totale* que nous présenterons en §16.1 p180. Le problème n°9 p84 est lié à l'utilisation de RMI. Toutefois, ce n'est pas véritablement une limitation : le modèle de programmation ProActive interdit le partage des objets passifs entre objects actifs. Cependant, lorsque deux objets d'une même bibliothèque sont rendus actifs, il n'est pas toujours possible de savoir s'ils ne partagent pas des objets passifs. C'est en particulier la cas lorsque la bibliothèque n'a pas été prévue pour être utilisée dans le cadre de programmation défini par ProActive. Une analyse de code statique ou dynamique pourrait apporter une solution au prix : d'une complexification des phases de développement dans le cas d'une analyse statique, et de déploiement dans celui d'une analyse dynamique. Enfin, les problèmes n°10 p84 à n°12 p84 sont liés au modèle objet actif lui-même ; nous en rediscuterons en §12.4.1.1 p134.

ProActive est facile à mettre en œuvre et à porter puisqu'il est 100 % Java. Sous les contraintes mentionnées ci-dessus, ProActive permet une distribution relativement transparente d'une application Java. C'est d'ailleurs le but premier du modèle objets actifs : distribuer (et donc paralléliser) un code séquentiel. La transparence joue donc un rôle fondamental. Cependant, en raison des problèmes énumérés ci-dessus, cette transparence peut être qualifiée de *transparence floue* : le comportement de l'application distribuée peut ne pas être le même que celui de sa version

[9] Ce problème est inhérent à la sémantique asynchrone et transparente des appels de méthodes sur un objet actif comme nous le verrons en §12.4.1.1 p134.

séquentielle. Nous verrons de toute façon que la transparence des appels asynchrones est un problème majeur en §16.1 p180 dans la partie §IV p123 consacrée à la concurrence.

Rendre un objet actif (distant) est dynamique dans ProActive. Cette opération entraîne un coût élevé en raison de la génération de classes à la volée[10]. L'équipe OASIS améliore ProActive de manière assez intensive, puisqu'un certain nombre de fonctionnalités ont été apportées : la communication de groupe [23], la sécurité, l'interopérabilité avec des environnements comme Globus ou Jini.

Enfin, ProActive possède une caractéristique essentielle : la mobilité. Les objets actifs peuvent être mobiles.

9.4 JavaParty

JavaParty [169] est dédié au développement d'applications parallèles pour le calcul intensif. Ce système est destiné aux *clusters* de stations reliées par un réseau rapide. JavaParty étend le langage Java pour permettre d'exprimer une distribution. Le développeur utilise cette extension pour distribuer ses applications.

Pour obtenir des performances optimales, une nouvelle implémentation de RMI [156] et de la sérialisation [165] a été réalisée. L'implémentation de RMI permet d'utiliser des protocoles de plus bas niveau que TCP/IP et d'exploiter les réseaux haut débit comme Myrinet [154]. Ces réalisations sont des bibliothèques que l'on peut charger à la place de celle fournie en standard avec le JDK.

JavaParty a lui aussi quelques limitations :

- les classes ne peuvent pas être téléchargées depuis le réseau, ce qui implique que toutes les stations doivent y avoir localement accès ;
- contrairement à la bibliothèque standard, on ne peut choisir son type de *socket* (via le *socket factory*) pour utiliser un protocole sécurisé comme SSL ;
- le réseau est supposé fiable (pas de `RemoteException`) ;
- contrairement à ce que l'on peut faire avec RMI, il n'est pas possible d'utiliser le protocole HTTP qui permet notamment d'exploiter des stations situées derrière un firewall.

Ces contraintes sont justifiées par le cadre d'utilisation de JavaParty qui est celui des *clusters* de stations. En effet, dans ce contexte, elles permettent d'éviter le stockage d'informations superflues et ainsi d'accélérer de manière significative les performances.

La distribution est assurée par le mot-clé `remote` ajouté au langage Java. La complexité de l'aspect distribué est complètement masquée à l'utilisateur (plus de

[10] Cependant, cette génération n'étant effectuée qu'une seule fois, le surcoût est minimum.

`RemoteException`) grâce à cette extension, mais l'accès aux sources est indispensable. Cette extension requiert la modification de certaines caractéristiques du langage d'origine. JavaParty redéfinit notamment les contrôles d'accès aux objets. Par exemple, les champs et les méthodes déclarés `private` sont transformés en `public`. La migration d'objet est assurée sous deux contraintes :

- aucune méthode de l'objet ne doit être traversée par une thread ;
- aucun objet ne doit être en cours de migration.

Pour une classe déclarée `Remote`, un pré-compilateur génère neuf classes 100 % Java. Ce nombre s'explique par la prise en compte de la sémantique du mot-clé `static` dans un cadre distribué. En effet, chaque classe `remote` est liée à un objet unique – sur l'ensemble des nœuds considérés – dont le rôle est de prendre en compte tous les accès aux variables et aux méthodes de classes.

JavaParty permet de développer des applications distribuées de manière transparente : il masque dans une certaine mesure les problèmes liés au réseau. JavaParty n'est pas dynamique, il faut avoir accès aux sources pour rendre un objet distant. Par ailleurs, l'application exécutée n'est plus tout à fait conforme à l'application écrite (modifications du code) ; cela pose des problèmes non négligeables lors du débogage.

Contrairement aux approches *Métacomputing*, JavaParty ne permet pas d'utiliser n'importe quelle ressource du réseau. Ce n'est pas, en effet, dans ce cadre qu'il a été prévu.

9.5 Do!

Do! [120, 121] est un environnement de programmation parallèle en Java. En définissant un modèle de développement d'applications parallèles génériques, il permet de distribuer une application implémentée dans ce cadre en changeant simplement le nom de bibliothèque (`import`). Un pré-compilateur se charge de rendre distants – au sens RMI – les objets marqués comme tels[11]. Les données sont organisées dans une `COLLECTION`. Elle peut être distribuée ou non selon le modèle d'exécution choisi (mémoire partagée, mémoire distribuée), et ce en reprenant les mécanismes de distribution que l'on trouve dans HPF par exemple (distribution par bloc, distribution cyclique, etc.). De ce point de vue, l'environnement Do! est naturellement bien adapté aux applications à parallélisme de données.

L'utilisation de l'héritage et de la réflexion permet d'utiliser la notion de *proxy* et de garantir l'accès transparent à un objet où qu'il soit, sans modification syntaxique. Toutefois, une méthode déclarée avec le mot-clé `final` ne pouvant être surchargée,

[11] En implémentant une interface spécifique : `ACCESSIBLE`.

cela n'est pas toujours possible, notamment pour distribuer des applications ou des bibliothèques dont on ne possède pas les sources. Par ailleurs, il n'est pas possible d'utiliser un accès direct aux champs d'un objet. Le *proxy* a en effet une copie locale du champ – par héritage – qui n'est pas synchronisée avec la valeur du champ de l'objet distant dont il a la charge. L'accès aux champs des objets doit se faire par un couple de méthodes (`get`/`set`). Ces points compliquent la réutilisation de bibliothèques dont on ne possède pas les sources.

Surtout, Do! impose un modèle de programmation parallèle qui ne permet pas nécessairement d'exprimer l'ensemble des problématiques concurrentes.

Do! ne semble plus être maintenu, la documentation de la seule version disponible semble en effet avoir été produite par la commande `javadoc` du JDK v1.0. Par ailleurs, le code source de cet environnement n'est pas disponible.

9.6 Les systèmes d'agents mobiles

Ces systèmes sont des cas particuliers de plates-formes distribuées. Nous avons vu qu'ils offrent de nombreuses fonctionnalités qui permettent, entre autres, de générer du parallélisme. Malheureusement, ils sont souvent lourds à mettre en œuvre car, de par leur nature, ils imposent une certaine rigidité pour la gestion de la sécurité.

L'aspect *agent mobile* est le plus important (sauf peut-être pour Voyager) et les objets utilisés doivent se conformer à un cadre particulier. Par exemple, dans le système des Aglets d'IBM, pour devenir mobile un objet doit nécessairement étendre la classe *Aglet*. Nous allons voir que l'utilisation des conteneurs actifs résout ce manque de dynamisme.

Enfin, généralement, les mécanismes de communication inter-agents sont relativement lourds et peu naturels. Par exemple, la communication fait intervenir un certain nombre d'APIs (modèle événementiel souvent) ; le développeur doit ainsi utiliser un certain nombre d'artifices syntaxiques pour effectuer un traitement suite à la réception d'un message. Ces mécanismes n'offrent donc pas vraiment de transparence dans la distribution, les développeurs étant habitués au paradigme classique d'appels de méthodes.

9.7 Les composants Enterprise JavaBeans

Les applications de commerce électronique, autrefois basées sur une architecture client/serveur, sont développées aujourd'hui suivant le modèle *multi-tiers*. Par exemple, une première couche nommée *couche présentation* est chargée de la présentation de pages HTML. Elle a besoin de données qu'elle peut demander à une

couche intermédiaire qui s'occupe de lui renvoyer les informations demandées. Cette deuxième couche fait appel à une troisième, par exemple une base de données qui contient l'ensemble des données de l'entreprise et exécute du code métier dont le résultat est renvoyé à la couche présentation. Finalement, la couche présentation crée une page HTML (ou autre) à la volée, en fonction des résultats fournis par la (les) couche(s) intermédiaire(s).

Un Enterprise JavaBean [194] est un composant d'une couche intermédiaire (*middleware*). L'idée étant de faire collaborer plusieurs composants pour effectuer un travail commun. La plate-forme EJB est entièrement dédiée à Java. Elle est basée sur le protocole RMI et permet donc de distribuer des objets à forte connotation *composants*. Certains services sont offerts comme la persistance ou le passage de messages. Toutefois, nous pouvons citer quelques contraintes dans le modèle :

- la distribution est statique : elle nécessite une phase déclarative et une phase de compilation, ce qui provient de l'utilisation du protocole RMI ;
- les appels de méthodes sont synchrones, mais les objets doivent être réentrants, deux clients pouvant potentiellement accéder à un même objet distant ; ce problème est lui aussi lié à l'utilisation du protocole RMI qui ne spécifie pas la politique utilisée lors d'appels concurrents sur un objet distant ;
- la déclaration de deux interfaces (`Remote` et `EJBHome`) et l'implémentation d'une classe dont certaines méthodes sont requises mais non détectées par le compilateur (injection de l'ensemble des méthodes de type `create<METHOD>` de l'interface `EJBHome` sur l'ensemble des méthodes de types `ejbCreate<METHOD>` de la classe d'implémentation) ;
- le déploiement de composants est relativement compliqué, requiert la génération d'un fichier de configuration du déploiement, d'une archive au format "jar" et le redémarrage des serveurs.

Enfin, cette plate-forme ne favorise pas le dynamisme : un composant EJB n'a pas de raison d'être en dehors du contexte des Enterprise JavaBeans, ce qui en réduit la réutilisabilité.

9.8 JavaSpace

JavaSpace est un service de la plateforme Jini [192]. Un tel service est un conteneur d'*entrées* (*entry*). Une entrée est un groupe *typé* d'objets exprimé sous forme de classe Java (elle implémente l'interface `net.jini.core.entry.Entry`). Une entrée peut être écrite dans un service JavaSpace, ce qui en crée une copie dans le conteneur.

Le service permet la recherche d'entrées en utilisant des *patrons* (*templates*). Deux types de recherche sont proposées : *read* et *take*. Une requête du premier type retourne soit une entrée associée au patron spécifié, soit une information indiquant

qu'aucune ne correspond. Une requête de type *take* suit le même mode opératoire que *read* mais si une entrée est trouvée, elle est retirée du conteneur.

Enfin, le conteneur peut avertir, selon un modèle événementiel, qu'une entrée correspondante à un patron donné a été écrite.

Toutes les opérations effectuées sur le conteneur sont transactionnelles. Toutes les entrées dans le conteneur sont associées à un *bail* (*lease*) : passé celui-ci, l'entrée est retirée du conteneur.

Ainsi, l'architecture JavaSpace est un conteneur distant d'objets. Cependant, les objets contenus dans un service JavaSpace sont passifs : aucune de leurs méthodes ne peut être exécutée par le service. C'est ici que la similitude avec nos conteneurs s'arrêtent : nos conteneurs sont *actifs*, ils peuvent invoquer les méthodes de leurs objets pour le compte d'un tiers (éventuellement distant).

9.9 Autres travaux

Nous trouvons dans la littérature de nombreux travaux qui peuvent être comparés à ceux que nous avons présentés et qui offrent le dynamisme que nous recherchons dans le développement d'applications distribuées orientées objets.

Par exemple, HORB [98] est un ORB (*Object Request Broker*) Java qui ne requiert pas de déclaration pour créer un objet distant. N'importe quel objet peut être compilé par un outil spécifique pour devenir distant. Par ailleurs, les appels de méthodes peuvent être asynchrones. Pour cela, une méthode `foo()` d'un objet distant doit être renommée en `foo_Asynch()` et les clients doivent utiliser les méthodes `foo_Request()` et `foo_Receive()` pour, respectivement, réaliser l'appel asynchrone de la méthode `foo()` et en récupérer le résultat.

Le projet RRMI (*Reflective Remote Method Invocation*) [203] est un autre exemple de projet qui se focalise sur le *dynamisme*. Contrairement au précédent qui utilise un compilateur ce dernier emploie la réflexion dans ce but – exactement comme nous le ferons. Il est cependant entièrement dédié aux appels distants et n'est pas conçu sur un modèle formel comme celui de nos conteneurs. En outre, il n'offre pas de mécanisme pour rendre l'utilisation de la réflexion transparente ce qui pose de nombreux problèmes comme nous le verrons en §16 p179.

Nous nous limiterons à ces deux exemples, la liste est en effet trop longue pour être présentée de manière exhaustive.

9.10 Bilan

Les exemples ci-dessus montrent que les outils de développement d'applications distribuées :

- sont dédiés à une architecture (*clusters* de stations et réseaux haut débit, Internet tout entier et protocole sécurisé, etc.) ;
- utilisent un type de méthode de développement (génération de code, transparent ou non transparent) ;
- utilisent un modèle de conception logicielle (objets actifs, `COLLECTION`, agents mobiles, composants EJB).

Par ailleurs, à part CentiJ (génération de code RMI), ProActive (héritage et redéfinition de méthodes) et Voyager (agrégation dynamique de facettes), beaucoup de ces outils sont statiques : un objet doit avoir été prévu pour être utilisé dans le cadre de l'outil. Une déclaration est nécessaire (`remote` pour JavaParty, `ACCESSIBLE` pour Do!, héritage d'une classe pour les systèmes d'agents mobiles et pour les composants EJB).

La transparence vis-à-vis du programmeur est proposée dans ProActive et Do! par la combinaison des mécanismes de *proxy* et d'héritage. Nous avons vu les limitations de cette approche (classes finales non héritables, méthodes finales non redéfinissables, problèmes d'accès aux champs, ...). JavaParty utilise une autre approche en modifiant le code pour qu'il soit conforme à une exécution dans un cadre distribué. Ces modifications rendent difficile le débogage d'une application. Les systèmes d'agents mobiles, pour leur part, ne proposent pas un modèle de programmation transparent. Voyager utilise un mécanisme assez élégant. Voyager utilise massivement le paradigme *conception par interfaces*. Tout objet implémente une interface. Ainsi, un proxy en implémentant l'interface de l'objet fournit la transparence voulue. Un compilateur (`igen`) fournit une interface aux objets qui n'en ont pas[12].

Nous allons voir comment l'utilisation du concept de conteneur actif permet d'apporter du dynamisme à moindre coût (en complexité conceptuelle et logicielle) en §10.5.1 p112.

9.11 Utilisation du modèle des conteneurs actifs

Cette section montre les utilisations potentielles du concept des conteneurs actifs. Nous ne détaillerons pas les deux utilisations triviales du concept qui exploitent seulement la singularité de la méthode `call()` :

[12] Attention, la présence de ce compilateur n'empêche pas le dynamisme. L'objet est bien distant. Seul le client doit utiliser l'interface générée pour communiquer avec l'objet.

- **Bus logiciel :** la méthode `call()` d'un conteneur actif distant permet de fait l'invocation de méthode à distance ; cela permet au concept d'être utilisé comme une couche de communication similaire à RMI ou à la couche de communication de CORBA ;
- **Parallélisme :** la métode `call()` d'un conteneur actif permettant l'exécution asynchrone de la méthode spécifiée en paramètre, le concept est utilisable pour exploiter la concurrence[13] d'une application.

9.11.1 Modèle mémoire

Le concept peut être utilisé comme le modèle mémoire d'une machine distribuée dans lequel tous les objets utilisés par un développeur sont des objets stockés (*stored objects*). Dans ce modèle, un objet stocké est un objet contenu dans un conteneur. La référence d'un tel objet est le couple

$$(activeContainer, key)$$

qui l'identifie de manière unique (pour peu que *activeContainer* soit un identifiant unique d'un conteneur actif). Dans un tel cadre, les méthodes du conteneur ont la signification suivante :

`void put(Object key, Object object)`	: crée un *stored object* ;
`void remove(Object key)`	: supprime un *stored object* ;
`Object get(Object key)`	: récupère une copie d'un *stored object* ;
`void call(Object key, String method, Object[] args, Result result)`	: invoque une méthode d'un *stored object*;

Évidemment, dans ce modèle, seuls les *stored objects* doivent pouvoir être manipulés, en aucun cas les objets réels.

Ces quatres méthodes trouvent une correspondance assez étonnante avec les primitives utilisées en Java pour la manipulation de la mémoire. En effet, il semble intuitivement possible de simuler une machine virtuelle Java à l'aide de ces quatres méthodes :

- la méthode `put()` permet de créer un objet au même titre que l'opérateur `new` du langage Java ;
- la méthode `call()` est sémantiquement équivalente à la notation pointée qui permet l'appel de méthode (`o.m(args)` correspond à `activeMap.call(key`$_o$`, "m", args)`) ;

[13] Dans un cadre distant, la concurrence engendre le parallélisme.

- le ramasse-miettes utilise la méthode `remove()` lorsqu'un objet n'est plus référencé ;
- la méthode `get()` est utilisée pour réaliser des copies (sérialisées) d'objets.

Évidemment, cela ne suffit pas pour implémenter le modèle mémoire Java [88] à base de conteneurs actifs mais ce modèle semble être un point de départ intéressant pour un tel projet.

Dans un cadre distribué, où seuls des *stored objects* sur des conteneurs actifs *distants* sont manipulés, la méthode `get()` permet d'implémenter :

- un mécanisme de cache : les objets, dont l'accès est en lecture seule (momentanément ou définitivement[14]), peuvent être rapatriés chez le client pour bénéficier de la localité ;
- un mécanisme d'équilibrage de charge mémoire, en tenant compte de la taille des objets ;
- un mécanisme d'équilibrage de charge CPU, en tenant compte du coût de leurs méthodes.

Nous verrons que notre implémentation permet assez facilement de développer des applications distribuées dans un tel modèle.

9.11.1.1 Résolution de noms dans le contexte du code mobile

L'utilisation des conteneurs actifs comme modèle mémoire offre au développeur un moyen relativement simple de gérer la résolution des noms dans un système d'agents mobiles (*cf.* §3.4.2 p35).

En effet, le référencement à distance est assuré par l'utilisation exclusive de références sur des stored objects[15]. La copie est assurée par la méthode `put()` du conteneur et le déplacement par les deux appels successifs `put()` et `remove()`. En développant dans le modèle mémoire conteneur actif, il doit être relativement naturel d'implémenter des mécanismes de résolution de noms.

9.11.2 Objets stockés multi-protocoles

La communication avec un *stored object* est assurée par le conteneur à travers sa méthode `call()`. Par conséquent, le protocole employé est celui utilisé par le conteneur. En adoptant le modèle de conteneur actif, nous pouvons fournir un mécanisme de communication directe qui utilise n'importe quel protocole comme RMI, IIOP, TCP, UDP, etc. Un tel mécanisme permet par exemple une communication chiffrée (TCP/SSL) entre *stored objects* sur un WAN comme Internet et en même temps,

[14] Objets dit *immutables*.

[15] Nous verrons la classe `StoredObjectReference` en §10.4 p111.

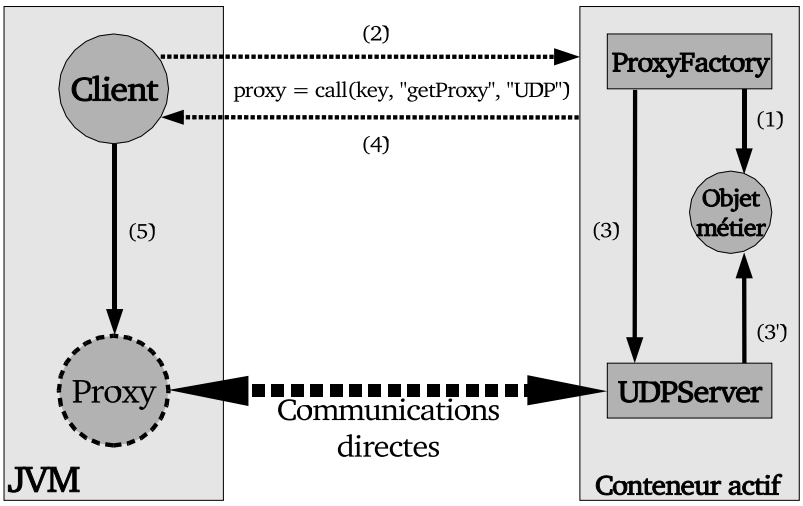


Fig. 9.3 – Objets stockés multi-protocoles : algorithme.

une communication ultra-rapide entre *stored objects* sur un LAN sécurisé en utilisant un protocole haute performance comme BIP/Myrinet.

Considérons l'algorithme suivant schématisé sur la figure Fig. 9.3 p93 : supposons qu'une fabrique de proxy (`ProxyFactory`) – qui a une référence locale sur un objet "métier" dont elle doit retourner des proxy – a déjà été insérée dans un conteneur actif (`put()`) (1). La méthode `getProxy()` est invoquée à distance via la méthode `call()` du conteneur actif (2) par le client. La fabrique instancie un nouveau serveur en fonction du protocole spécifié si nécessaire[16] (ici `"UDP"`) (3) et la référence locale à l'objet métier lui est fournie (3)'. Le proxy associé au serveur est renvoyé au client (4). Celui-ci ayant une référence sur le proxy (5), peut alors communiquer directement avec son objet métier, sans passer par le conteneur et en utilisant le protocole désiré (6).

Nous appelons ce mécanisme *Multi-Protocols Stored Objects* (MPSO) et nous l'avons proposé au domaine des transmissions sécurisées sur PDA dans un article intitulé *Just In Time Security* (JITS) [51].

9.11.3 Déploiement d'applications

Le conteneur actif facilite le déploiement d'applications à grande échelle. En effet, le concept de conteneur est très simple : il garantit des implémentations peu gourmandes en ressources (mémoires et CPU) facilitant ainsi son adoption à grande échelle.

[16]Une telle instance peut déjà avoir été créé au préalable par un appel similaire d'un autre client.

Il faudra toutefois gérer les problèmes de sécurité : nous avons vu que la migration de code peut poser des problèmes. Par ailleurs, dans une zone sécurisée comme un laboratoire ou un réseau d'entreprise, le conteneur peut éventuellement remplacer les démons de type rsh, ou ssh (pour peu que la sécurité des conteneurs soit assurée).

Déployer une application revient à :

- instancier des fragments de l'application sur chaque machine et à lancer l'application distribuée qui utilise l'ensemble de ces fragments pour résoudre un problème donné (application parallèle par exemple, mais pas seulement) ;
- installer une application sur n machines afin de la rendre disponible aux utilisateurs.

Les conteneurs permettent de résoudre ces deux problèmes qui habituellement demandent des solutions distinctes. Dans le premier cas, l'application insère dans chaque conteneur les fragments de son application et les utilise par l'intermédiaire de la méthode `call()`[17]. Dans le deuxième cas, on peut supposer qu'il existe une application d'installation sur chaque conteneur (si ce n'est pas le cas, la première qui a besoin de cette fonctionnalité l'insère) ; l'administrateur demande alors à chacun de ces *stored objects* d'installer son application (l'utilisation d'une communication groupée peut s'avérer judicieuse).

9.11.4 Extension d'applications

L'ajout de *plugin* est une fonctionnalité très répandue dans les applications actuelles. Cette fonctionnalité nécessite généralement d'être conçue dans l'application. Souvent, il faut la redémarrer. Par exemple, un serveur HTTP type Apache doit-être redémarré, et même recompilé pour supporter une fonctionnalité particulière (module PHP, Servlet, etc.). L'utilisation des conteneurs apporte cette fonctionnalité aux applications qui ne l'ont pas prévue. Le plugin peut être :

- un client du conteneur actif dans lequel réside l'application ; il communique avec son application par l'intermédiaire du conteneur ;
- inséré dans le conteneur pour l'étendre ; les clients de l'application peuvent communiquer avec le plugin par l'intermédiaire du conteneur.

Enfin, pour les applications qui utilisent le modèle de conteneur pour leur communication (utilisation de *stored objects* exclusivement), un plugin peut être un objet quelconque inséré après coup dans n'importe quel conteneur.

[17] Ou en utilisant une communication directe comme expliqué en §9.11.2 p92

Chapitre 10

Implémentation en Java: JACOb

Nous avons fait émerger le concept de *conteneur actif* dans le chapitre précédent. Notre modélisation de ce concept en π-calcul a été donnée en §D p313 et nous avons vu que ce concept n'est pas limité à l'utilisation dans les plate-formes d'agents mobiles. Ce chapitre va exposer une implémentation de ce concept en Java que nous avons nommée JACOb pour *Java Active Container of Objects*.

10.1 Conception par interfaces

Nous avons souhaité concevoir notre plate-forme à base de conteneurs actifs en utilisant le paradigme *design by interfaces*. Celui-ci permet de définir une spécification de l'architecture que les implémentations doivent suivre.

En Java, la notion de conteneur est fournie dans le paquetage généralement appelé *paquetage collection*. En réalité, aucun *package*[1] n'est dédié à cet effet. Le paquetage collection est distribué au sein du *package* `java.util`.

Ce dernier possède une interface `java.util.Map` qui est très similaire à l'interface de notre conteneur[2]. Aussi, nous avons décidé de créer l'interface `ActiveMap` qui hérite de l'interface `java.util.Map` en y ajoutant la méthode `call()`. Cette dernière méthode prend en argument :

- la clé du *stored object* ;
- la méthode à invoquer sur le *stored object* ;
- les arguments de cette méthode dans un tableau d'`Object` ;

[1]Nous distinguerons le terme "paquetage" qui prendra le sens habituel "collection de classes" ou "bibliothèque de classes", du mot-clé en Java *package* qui a nécessairement une existence physique. Ainsi, le paquetage JFC – aussi appelé Swing – dédié au développement d'interface graphique portable, est en réalité composé d'un ensemble de *packages* : `java.awt` et `javax.swing.*`.

[2]Les différences se situent au niveau des appels `put()` et `remove()` qui retournent `java.lang.Object`.

- un moyen de récupérer le résultat ;

La spécification de la méthode à invoquer mène naturellement à l'utilisation du mécanisme de *réflexion* proposé en Java depuis la version 1.1. Aussi, le deuxième argument sera un `java.lang.reflect.Method`[3]. Cette particularité impose de disposer de la classe dont on veut invoquer la méthode, même si l'appel est réalisé à distance. En utilisant une simple chaîne de caractères, cette contrainte aurait pu être levée. Cependant, nous considérons que dans un paradigme à base d'appel de méthode, disposer de la classe de l'objet que l'on veut utiliser est une contrainte naturelle. Par ailleurs, nous verrons en §16 p179 que l'utilisation de la méthode `call()` n'est pas le moyen recommandé pour se servir des conteneurs. Si cette contrainte est considérée comme trop importante, l'utilisateur doit se tourner vers des solutions plus génériques, comme *Java Message Service* [147] par exemple.

L'appel de la méthode spécifiée dans `call()` doit être asynchrone. Cela implique que son résultat ne peut être connu au retour de la méthode `call()`. Un mécanisme particulier doit donc être employé pour la récupération du résultat d'un appel asynchrone. Ces mécanismes sont décrits dans la partie qui traite de la concurrence en §14.4 p152. Nous utilisons un mécanisme *d'attente anticipée* – un objet *futur* – qui permet au client de scruter ou d'attendre un résultat.

Pour récupérer cet objet *futur*, il y a deux possibilités :

- il est créé et renvoyé par la méthode `call()` ;
- il est passé en argument à la méthode `call()` ;

Le choix va être déterminé par les différentes utilisations possibles de notre interface `ActiveMap`. En particulier, dans un cadre distribué, des implémentations distantes de cette interface seront utilisées. Or, un retour de méthode implique nécessairement un envoi de message, tandis que la spécification d'un argument permet au client de délibérément utiliser une référence particulière (`null` par exemple) que l'implémentation de l'`ActiveMap` interprétera comme : ne rien renvoyer. C'est le mécanisme *one-way messaging* utilisé dans Voyager par exemple. Par conséquent, la deuxième approche sera utilisée : l'objet *futur* devra être créé par le client et passé en paramètre à notre méthode `call()`. L'interface de cet objet *futur* doit permettre – nous l'avons dit – la scrutation et l'attente du résultat. Aussi, nous proposons l'interface §10.1 p97[4].

[3]Cela conduira à quelques soucis liés au fait que la classe java.lang.reflect.Method n'est pas sérialisable (bug n°4171142). Plusieurs solutions ont été proposées et implémentées. Celle retenue est une classe sérialisable `SerializableMethod` qui possède les mêmes méthodes que la classe `java.lang.reflect.Method`. Nous verrons les conséquences de cette implémentation en §15.2.1.1 p162

[4]Nous verrons en §15.1.1 p158 que cette interface n'existera plus après notre étude de la concurrence au profit d'un découpage plus fin.

```
1  public interface Future {
2  Object waitForResult() throws Throwable, InterruptedException;
3  Object waitForResult(long timeout)
4          throws Throwable,
5          InterruptedException,
6          TimedOutException;
7
8  boolean isResultAvailable();
9  Object getReturnedResult() throws Throwable;
10
11 // Utilisé par le serveur lorsque l'appel de méthode
12 // est terminé
13 void setResult(Object result, Throwable throwable);
```

Prog. 10.1 – L'interface Future (non définitive)

Les méthodes `waitForResult()` permettent d'attendre la fin de l'appel asynchrone et de récupérer le résultat. Si la méthode invoquée a levé une exception, elle le sera à nouveau, ce qui permet au client de la gérer. La méthode `isResultAvailable()` permet la scrutation du résultat, et la méthode `getReturnedResult()` sa récupération. L'appel à cette dernière est indéfini si l'invocation effective n'est pas terminée, c'est-à-dire si la méthode `isResultAvailable()` ne retourne pas `true`.

Finalement, l'interface de notre *ActiveMap* est donnée[5] en Prog. 10.2 p97.

```
1  public interface ActiveMap extends Map {
2     void call(Object key,
3               java.lang.reflect Method method,
4               Object[] args,
5               Future future);
6  }
```

Prog. 10.2 – L'interface ActiveMap (non définitive)

10.1.1 Implémentation locale

Les interfaces ont été définies et doivent être implémentées. Notre réalisation de l'interface `ActiveMap` nommée `ActiveMapImpl` hérite directement de la classe `java.util.HashMap` afin de réutiliser le code du conteneur (`put()`, `get()`, et `remove()`, entre-autres).

[5]Cette interface va évoluer après notre étude de la concurrence en §IV p123.

L'implémentation de la méthode `call()` est relativement simple[6] : on trouve l'objet associé à la clé spécifiée et on alloue une thread pour qu'elle effectue l'appel de méthode spécifié et l'enregistrement du résultat en utilisant la méthode `setResult()` de l'objet *futur* passé en paramètre.

Enfin, l'implémentation de l'interface `Future` est relativement triviale. Il faut simplement prendre garde à réveiller les threads bloquées dans les méthodes `wait*()` de l'objet *futur* lorsque le résultat est mis à jour par le conteneur via la méthode `setResult()` mentionnée plus haut.

10.2 Prise en compte de l'aspect distant

La réalisation précédente est une implémentation locale d'un conteneur actif. Les conteneurs ont pour vocation d'être distribués : c'est essentiellement là que réside leur intérêt. Le concept de conteneur actif doit faciliter le développement d'applications distribuées en rapprochant la programmation distribuée – réputée difficile – du cadre bien connu de la programmation locale (et séquentielle). Ce rapprochement est jugé impossible par Jim Waldo, Geoff Wyant, Ann Wollrath et Sam Kendall dans leur article intitulé : "A Note On Distributed Computing" [221] – à l'origine de Java-RMI – car l'aspect distant apporte des problèmes nouveaux qui n'ont pas d'équivalent dans le domaine local. Cet article identifie quatre caractéristiques majeures existant nécessairement dans un système distribué : la latence, l'accès mémoire, les pannes partielles et la concurrence. Afin d'unifier la programmation distribuée et la programmation séquentielle, il est indispensable d'unifier ces quatre caractéristiques, ce qui est très compliqué pour les deux premières et jugé impossible pour les deux dernières.

Si des critiques récentes ont été apportées sur cet article [189], nous avons aussi vu les difficultés que pose Java-RMI (*cf.* §9.1 p79). Nous pensons donc qu'un rapprochement significatif peut être effectué entre les deux mondes de la programmation distribuée et de la programmation séquentielle. Le concept de conteneur actif va faciliter ce rapprochement, mais nous verrons qu'il ne suffira pas. C'est pourquoi, la partie §IV p123 sera nécessaire.

Nous allons donc maintenant présenter les quatre caractéristiques fondamentales des applications distribuées. Nous montrerons ensuite les rapprochements qu'il est possible d'effectuer en utilisant le concept de conteneur actif et plus particulièrement notre implémentation : JACOb.

[6] Nous complexifierons nettement cette implémentation lorsque nous aurons étudié les problèmes liés à l'asynchronisme en §15.2.3.2 p170.

10.2.1 Latence

La latence est définie par le délai entre l'invocation d'une méthode et son exécution effective. La latence introduite par le réseau dans un appel de méthode à distance peut être de plusieurs ordres de grandeur supérieure à celle d'un appel local. Ce problème est très connu dans le domaine du parallélisme. On le résout par un placement efficace des différents processus communicants. Il peut être effectué au lancement de l'application (déploiement) après analyse statique du graphe de tâches, à l'exécution par analyse dynamique du comportement de l'application et migration de processus (ce qui pose des problèmes non-négligeables comme par exemple la récupération de l'état global de l'application) ou les deux. Ces mécanismes sont assez simples à implémenter en utilisant le concept de conteneur actif : un programme d'analyse (statique, dynamique ou les deux) d'applications orientées objets distribuées place les instances d'objets dans les conteneurs actifs appropriés de telle sorte que les communications soient globalement réduites à leur minimum. Des recherches dans cette direction sont menées dans notre équipe, avec la notion d'objets *séparables* [48].

10.2.2 Accès mémoire

Le problème d'accès à la mémoire dans un cadre distribué est lié au fait qu'un accès distant ne peut être considéré comme un accès local. En effet, lors d'une invocation de méthode à distance, les paramètres doivent être sérialisés pour être transférés sur la machine distante : le paradigme utilisé dans ce cas est le *"passage par copie"* et non l'habituel *"passage par référence"*. Un code peut donc utiliser la même syntaxe pour exprimer deux paradigmes d'appel de méthode différents. RMI a ce problème. Pire, selon le type du paramètre, le passage est par référence (cas des objets de type `java.rmi.Remote`) ou par copie (objets de type `java.io.Serializable`). Enfin, certains paramètres ne peuvent être passés en argument d'une méthode distante (tous les autres objets) alors qu'ils sont parfaitement compatibles avec le type du paramètre déclaré par la méthode (par exemple une méthode d'un objet RMI `void foo(java.lang.Runnable runnable)` à qui l'on passe en argument une instance de classe non sérialisable comme `java.lang.Thread` mais implémentant l'interface `java.lang.Runnable`).

L'utilisation du conteneur actif comme modèle mémoire résout ce problème en forçant le développeur à clairement exprimer le paradigme qu'il souhaite utiliser pour son appel de méthode, restant fidèle au cas local :

- chaque référence utilisée dans le programme doit être une référence sur un *stored object* (*cf.* §9.11.1 p91 et §10.4 p111), et les communications entre *stored*

objects doivent être effectuées par l'intermédiaire de la méthode `call()` exclusivement ; cela assure un paradigme de passage des paramètres *par référence* ;
- lorsqu'un *passage des paramètres par copie* est souhaité, la méthode `put()` des conteneurs actifs doit être utilisée – exactement comme l'on utilise la méthode `clone()` dans le même but localement.

10.2.3 Panne partielle

La panne partielle est l'une des caractéristiques principales des architectures distribuées. Dans le cas local, une panne est soit totale, soit détectable (par le système d'exploitation par exemple). Dans le cas distant, un composant (un lien du réseau, un processeur) peut tomber en panne pendant que les autres continuent de fonctionner correctement. Pire, il n'y a pas de composant central capable de détecter les pannes et d'en informer les autres, ni d'état global qui décrit ce qui s'est exactement passé. Par exemple, lorqu'un lien réseau est coupé, un appel de méthode précédemment invoqué peut s'être terminé normalement et avoir provoqué un effet de bord dans une base de données, ce qui n'est pas le cas si le processeur est tombé en panne.

Notre modèle ne spécifie pas comment les pannes doivent être gérées. D'ailleurs, l'aspect distant n'apparaît absolument pas dans l'interface `ActiveMap` : c'est à l'implémentation de fournir les mécanismes appropriés.

Nous allons étudier deux mécanismes.

10.2.3.1 Gestion par appel de méthode

Dans l'approche *par appel de méthode*, le client, à chaque invocation à distance, vérifie que l'appel s'est bien terminé. Le standard Java-RMI utilise ce mécanisme en forçant le client à encadrer ses appels distants par un bloc `try/catch`.

Pour cela, chaque méthode d'un objet distant doit[7] être déclarée comme susceptible (`throws`) de lever l'exception contrôlée `RemoteException`. Cette exception est levée par les classes du paquetage RMI lorsqu'un problème lié à l'aspect distant survient : sérialisation/désérialisation des arguments, communications ou chargement dynamique de classes par exemple. Si cette démarche permet, dans une certaine mesure[8], de s'assurer que les appels distants ne sont pas utilisés comme des appels locaux (la sémantique du passage de paramètres est différent comme nous l'avons

[7] Vérifié par le compilateur `rmic`.

[8] Il en effet assez courant de déclarer la méthode appelante "levant l'exception `RemoteException`" ou d'encadrer l'ensemble des appels distants par un seul `try/catch` ce qui ne permet plus de distinguer la différence syntaxique entre un appel local et un appel distant.

vu en §10.2.2 p99), elle complexifie quelque peu le développement pour plusieurs raisons essentielles.

Tout d'abord, la gestion pour chaque appel de méthode des pannes éventuelles liées à l'aspect distant est assez complexe à mettre en œuvre. Il faut en effet mélanger les exceptions "métier" liées à la sémantique de l'appel de méthode avec les exceptions "réseau" liées à la nature distante de l'appel. Par ailleurs, la gestion d'une panne réseau est généralement réalisée de façon globale pour l'ensemble de l'application : continuation sur un serveur secondaire, avertissement de l'administrateur par courrier éléctronique, redémarrage de serveur, etc.

Ensuite, cette approche n'est absolument pas justifiée dans le cadre de réseaux fiables à très haut débit par exemple dans lesquels une panne, même partielle, est considérée comme fatale et donc totale.

Finalement, cette démarche interdit le polymorphisme. Il n'est par exemple pas possible d'étendre une interface non-RMI et de la rendre RMI. En effet, les exceptions déclarées font partie de la signature des méthodes. Considérons par exemple l'interface `java.util.Observer` contenant la méthode suivante :

```
void update(java.util.Observable obs, java.lang.Object o);
```

L'implémenter en utilisant RMI n'est pas possible directement. Nous pourrions définir notre propre interface `RMIObserver` qui hérite de cette dernière et surcharge la méthode précédente avec une autre méthode qui déclare lever l'exception `RemoteException`. Mais cela n'est pas possible : les deux méthodes ont les mêmes arguments ce qui ne permet pas de les différencier, ni à la compilation, ni à l'exécution. Le seul moyen[9] est donc d'utiliser le *pattern Adapter* [79]. On crée :

- l'interface `RMIObserver` mentionnée plus haut qui hérite de `java.rmi.Remote` au lieu de `Observer` ;
- l'implémentation "métier", une classe qui hérite de `java.rmi.server.UnicastRemoteObject` (et qui implémente l'interface `RMIObserver` évidemment) ;
- une classe *adapter* qui implémente l'interface `Observer` en utilisant le *stub* RMI de type `RMIObserver` pour transférer les appels de méthodes vers la classe "métier" comme nous l'avons vu sur la figure Fig. 9.1 p80.

Ce pattern sera massivement utilisé pour l'implémentation de JACOb au-dessus de RMI : nous le découvrirons en §10.3.1 p105.

Évidemment, cette solution ne fait que déplacer le problème : la classe *adapter* doit prendre en compte les exceptions potentiellement générées par le *stub*. Elle doit donc offrir un mécanisme qui permet à l'utilisateur de gérer ces exceptions.

D'une manière plus générale, certains [111, 94] considèrent que les exceptions

[9]Du moins, le seul que nous ayons trouvé jusqu'à présent.

contrôlées fragilisent un système, augmentent la taille du code, et cause des tourments aux développeurs.

10.2.3.2 Gestion globale événementielle

L'autre moyen de prendre en compte les exceptions liées à l'aspect distant d'un appel de méthode est de séparer les exceptions "métiers" (*cf.* paragraphe précédent) des exceptions réseaux. Les premières doivent être gérées naturellement, comme toutes les autres exceptions, de manière locale pour chaque méthode tandis que les secondes doivent être gérées de façon globale (ou pas du tout dans le cas d'un réseau fiable).

10.2.3.3 Gestion des pannes dans JACOb

JACOb permet d'utiliser un "algorithme central" capable de décider, en fonction de l'exception réseau, de la méthode invoquée, des paramètres, de l'objet distant impliqué, si le code qui a généré cette exception doit continuer son exécution comme si de rien n'était – en utilisant un mécanisme qui permet de remettre les choses dans un état cohérent – ou s'il doit au contraire être interrompu par une levée d'exception.

Pour cela, JACOb utilise une gestion hiérarchisée des exceptions réseaux : un gestionnaire d'évènements est utilisé de manière globale pour chaque objet *primitif distant*[10]. Ce gestionnaire va avoir les moyens de déléguer la gestion de l'exception réseau au client de l'appel. Il peut aussi ne rien gérer du tout (cas d'un réseau fiable).

Ainsi, JACOb définit l'interface `Remote` en PROG. 10.3 p102 que doit implémenter toute classe *primitive distante*.

```
1 public interface Remote{
2         ExceptionHandler getExceptionHandler();
3         void setExceptionHandler(ExceptionHandler);
4 }
```

PROG. 10.3 – L'interface `Remote` dans JACOb pour la gestion des exceptions réseaux.

Par cette interface un objet primitif distant est nécessairement associé à un *gestionnaire d'exceptions réseaux* : une instance de classe qui implémente l'interface `ExceptionHandler`. Cette interface est présentée en §10.4 p103.

[10]Un objet est *primitif distant* s'il est directement accessible à distance sans passer par un conteneur actif. Un certain nombre de ces objets sont implémentés dans JACOb : par exemple un conteneur actif distant utilisant le protocole RMI (`RMIActiveMap`).

```
1 public interface ExceptionHandler{
2     Object handleException(Object exceptionInfo);
3 }
```

Prog. 10.4 – L'interface `ExceptionHandler` associée à toute classe *primitive distante* dans JACOb pour la gestion des exceptions réseaux.

Cette interface définit la méthode `handleException()` qui permet d'avertir le gestionnaire qu'une exception est survenue. L'argument passé à cette méthode est laissé libre[11]. Il peut donc être une référence sur l'exception levée mais aussi un objet donnant des informations sur la méthode qui a levé l'exception.

Les objets *primitifs distants* sont des instances de classes qui héritent de la classe abstraite `AbstractRemote` qui implémente essentiellement[12] l'interface `Remote` vue précédemment, illustré par la partie nommée *Partial failures classes* du schéma UML Fig. 10.1 p106. Surtout, cette classe spécifie que l'argument de la méthode `handleException()` du gestionnaire d'exception doit s'attendre à recevoir une instance de la classe `ExceptionInfo` qui fournit, outre une référence sur l'exception qui a été levée, la méthode qui a levé l'exception – par l'intérmédiaire de la réflexion, et les arguments de cette méthode. Le comportement du gestionnaire, en fonction de ces informations, peut donc être :

- de gérer l'exception de manière transparente pour le client : il lui suffit de retourner un objet représentant le résultat de la méthode qui a levé l'exception (résultat par défaut, résultat en cache ou résultat d'un appel mirroir par exemple) ;
- de déléguer la gestion de l'exception au client en retournant une valeur particulière (`REJECT`) : le client recevra une exception non-contrôlée `UnHandledException` qu'il a les moyens de récupérer s'il le souhaite (bloc `try/catch` classique).

Entre ces deux extrêmes, le gestionnaire peut par exemple envoyer un avertissement à l'administrateur de l'objet distant et/ou redémarrer le serveur tout en retournant la valeur `RejectExceptionHandling.REJECT` demandant au client de gérer au mieux l'exception qui vient d'être levée.

[11] En réalité, l'interface `ExceptionHandler` n'est pas dédiée au problème des pannes partielles et ne fait donc pas partie de l'ensemble de classe JACOb. Cette interface appartient à un paquetage utilitaire nommé `mandala.util`.

[12] Elle contient aussi les méthodes d'accès au système de journal (*log*), utilisé pour l'administration et le débogage.

10.2.4 Concurrence

Les objets distants doivent généralement faire face à des appels de méthodes conncurrents potentiellement de plusieurs sources différentes. Généralement, cela implique qu'un certain nombre de threads sont utilisées dans cette gestion, souvent par l'intermédiaire d'un *thread pool*.

Par conséquent, les objets non *thread-safe* ne peuvent devenir distants directement sans mécanisme de séquentialisation des appels. Le modèle précise que l'appel `call()` est asynchrone, ce qui semble empêcher l'insertion d'objets non *thread-safe* dans un conteneur. L'aspect asynchrone est entièrement pris en charge par un paquetage séparé nommé RAMI et que nous présenterons au chapitre §14 p147.

10.3 Implémentation des classes primitives distantes

Nous avons vu l'interface `ActiveMap` qui définit le conteneur actif et son implémentation locale `ActiveMapImpl`. L'implémentation distante de cette interface est une classe *primitive distante*. Nous utiliserons des objets fragmentés de type *proxy/serveur* pour toutes nos implémentations de classe *primitive distante*. Si l'interface principale dans JACOb est celle qui définit le conteneur actif, `ActiveMap`, elle hérite de l'interface `java.util.Map`. Les implémentations doivent donc définir l'ensemble des méthodes de `java.util.Map` en plus de la méthode `call()` – y compris les implémentations distantes, ce qui va engendrer quelques difficultés. En effet, l'interface `java.util.Map` définit, par exemple, la méthode `keySet()` qui retourne une "*vue*" sous forme d'ensemble – au sens `java.util.Set` du terme – des clés contenues dans le conteneur. Les modifications effectuées sur cet ensemble doivent être répercutées sur le conteneur dont il est issu. Par conséquent, dans un cadre distant, la méthode `keySet()` ne doit pas retourner une copie sérialisée de l'ensemble initialement renvoyé mais bel et bien un *proxy distant* sur cet ensemble. Il faut donc implémenter l'interface `java.util.Set` dans une classe *primitive distante*. Comme `java.util.Set` hérite de `java.util.Collection`, il est raisonnable d'implémenter aussi de manière distante l'interface `java.util.Collection`. Comme l'ensemble des classes et des interfaces du paquetage *collection* sont étroitement liées, en utilisant le même raisonnement, la quasi-totalité du paquetage doit-être implémentée de façon distante.

La gestion des pannes est assurée dans JACOb grâce à l'interface `Remote`[13] que

[13] Définie par le code §10.3 p102.

doit implémenter toute classe *primitive distante*. Aussi, toutes les interfaces du paquetage *collection* trouvent une classe associée dans JACOb dont le nom est composé de celui de son interface mère préfixé de la chaîne de caractères `Remote`. Elles héritent à la fois d'une interface du paquetage *collection* et de l'interface `Remote`. On trouve entre autres les interfaces `RemoteMap`, `RemoteCollection` et `RemoteIterator`. La partie nommée *User classes* du schéma UML Fig. 10.1 p106 illustre les liens de l'interface `RemoteMap` avec `java.util.Map` et avec la partie nommée *Partial failures classes* (déjà vue en §10.2.3.3 p102).

10.3.1 Implémentation RMI directe

Dans un premier temps, nous avons "rapidement" développé une version en utilisant le protocole RMI. Sans entrer dans les détails, nous allons étudier l'implémentation RMI d'une des interfaces *primitives distantes* : `RemoteMap`.

La partie métier est assez simple : on définit l'interface `RMIMapOp` que l'on implémente dans une classe `RMIMapImpl`. Cette classe utilise la composition d'objets plutôt que l'héritage : n'importe quelle *map* pourra ainsi être rendue distante. Les méthodes de cette classe font simplement appel à l'instance que l'on rend distante. On utilise le pattern *Adapter* pour adapter l'interface non-RMI de `java.util.Map` en une interface RMI : `RMIMapOp` comme l'illustre le schéma §10.1 p106. Le client d'une *map* distante utilisera un proxy qui implémente l'interface `RemoteMap`. Ce proxy ne peut évidemment pas être un stub RMI puisque les méthodes d'un stub RMI doivent déclarer lever l'exception `RemoteException`. On utilise donc à nouveau le pattern *Adapter* pour rendre le stub RMI qui implémente l'interface `RMIMapOp` compatible avec l'interface `RemoteMap`.

Il faut prendre garde à l'implémentation : la méthode `keySet()` par exemple doit rendre distante la collection renvoyée par l'instance composée, et retourner un *proxy*. Cela suppose donc une forte corrélation entre les différentes classes *primitives distantes*. Il faut en effet que la collection retournée soit rendue distante de manière *dynamique*. Au niveau utilisateur, seul le *proxy* est visible (déclaré `public`) et il peut être intéressant de lui fournir cette fonctionnalité. Aussi, elle sera implémentée dans la classe *proxy*. Donc, toutes les classes *proxy* contiennent une méthode `newServer()` qui prend en argument une classe du paquetage *collection*, la rend distante et retourne un *proxy*.

On distingue dans l'architecture de classes présentée par le schéma UML §10.1 p106 les différentes couches impliquées : la couche "utilisateur", la couche de gestion des pannes et l'implémentation RMI proprement dite.

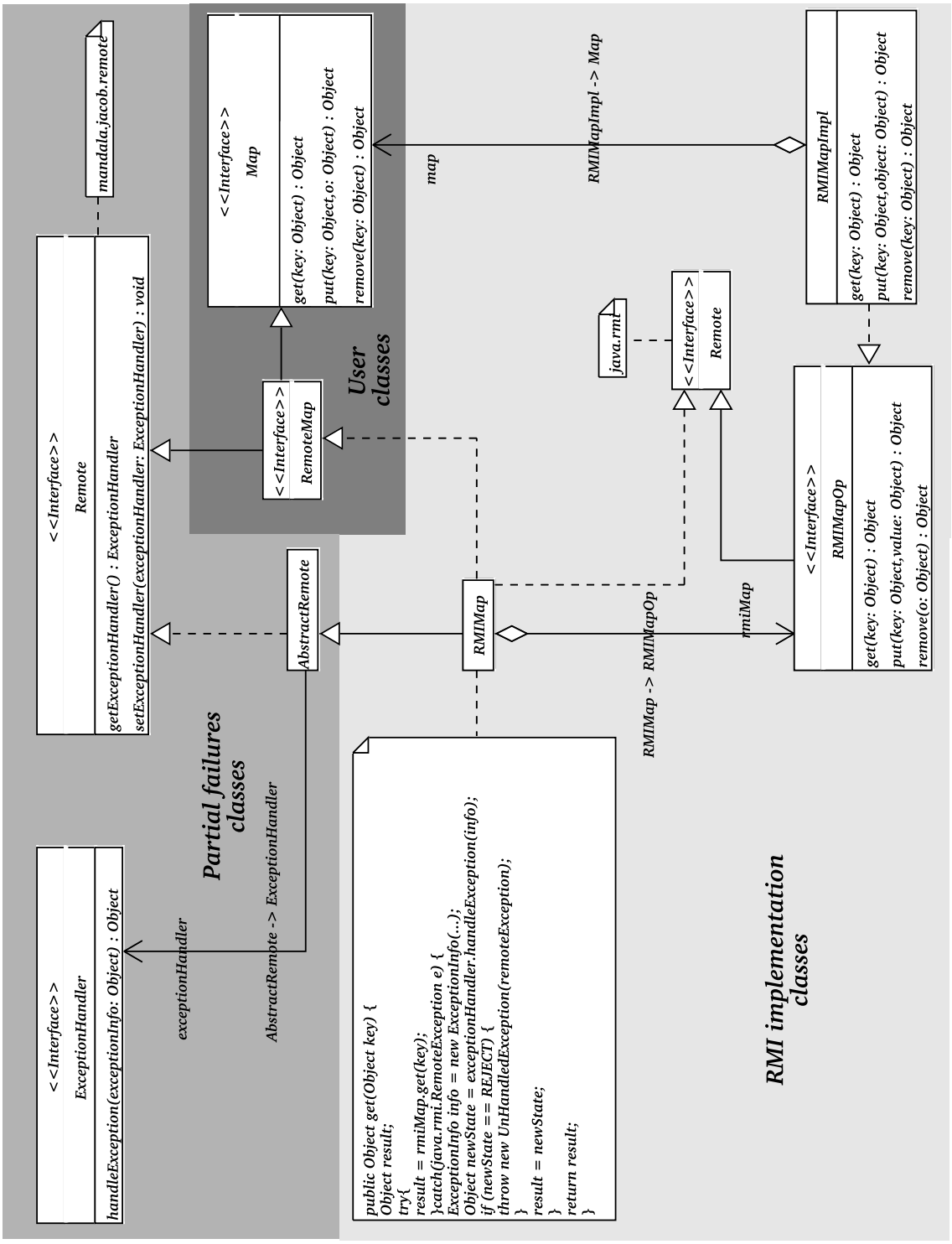


FIG. 10.1 – Architecture de classe pour l'implémentation RMI de `RemoteMap`

10.3.2 Implémentation générique

La première implémentation directe utilisant RMI souffre de deux défauts majeurs inhérents au protocole RMI :

- **RMI est lent :** c'est un problème récurrent que tentent de résoudre un certain nombre de travaux [156, 131, 117], dont ceux de notre équipe [50] ;
- **RMI est inadapté :** certains protocoles de communication n'emploient pas le *paradigme socket* généralement utilisé sous IP (BIP [174]/Myrinet [154], LAPI [77] et JToe [90] en sont quelques exemples).

Par conséquent, JACOb fournit un ensemble de classes dans la paquetage `mandala.jacob.remote.gpf` qui facilite l'implémentation des classes primitives distantes : c'est le *Generic Protocol Framework*.

L'architecture du GPF permet une implémentation distante rapide de JACOb quelque-soit l'API sous-jacente utilisée. Sans entrer dans les détails, une nouvelle implémentation qui utilise le GPF doit :

- implémenter l'interface `Exporter` qui permet de rendre distant n'importe quel *objet primitif* de JACOb (`ActiveMap`, `Map`, `Collection`, etc.) ;
- implémenter l'interface `Client` qui permet d'invoquer à distance une méthode d'un objet primitif distant ;
- hériter de chaque classe abstraite `Remote*Proxy` afin qu'elle utilise les implémentations des interfaces `Client` et `Exporter` pour la communication.

Ainsi, il est relativement simple d'implémenter JACOb afin d'utiliser un nouveau protocole de communication.

JACOb propose trois implémentations du GPF : UDP, TCP et RMI. Le schéma UML §10.2 p108 présente l'architecture de classes pour la version UDP. On distingue les différentes couches impliquées : la couche "utilisateur", la couche de gestion des pannes, la couche du GPF et enfin l'implémentation UDP proprement dite.

Si l'implémentation TCP semble être la plus naturelle, elle a pourtant été négligée au profit des deux autres : RMI et UDP. L'intérêt d'une implémentation RMI est de mesurer l'*overhead* engendré par l'utilisation du GPF. Une version UDP est justifiée pour deux raisons :

Le faible coût du protocole UDP par rapport au protocole TCP : le protocole UDP est beaucoup moins lourd que le protocole TCP puisqu'il rajoute seulement la notion de port au protocole IP (*cf.* les *Request For Comments* numéro 791 (IP) [171], 768 (UDP) [170] et 793 (TCP) [172]) ;

L'adéquation à un mode de communication asynchrone sans panne partielle : UDP est un mode *non connecté* dans lequel il n'y a pas de garantie sur

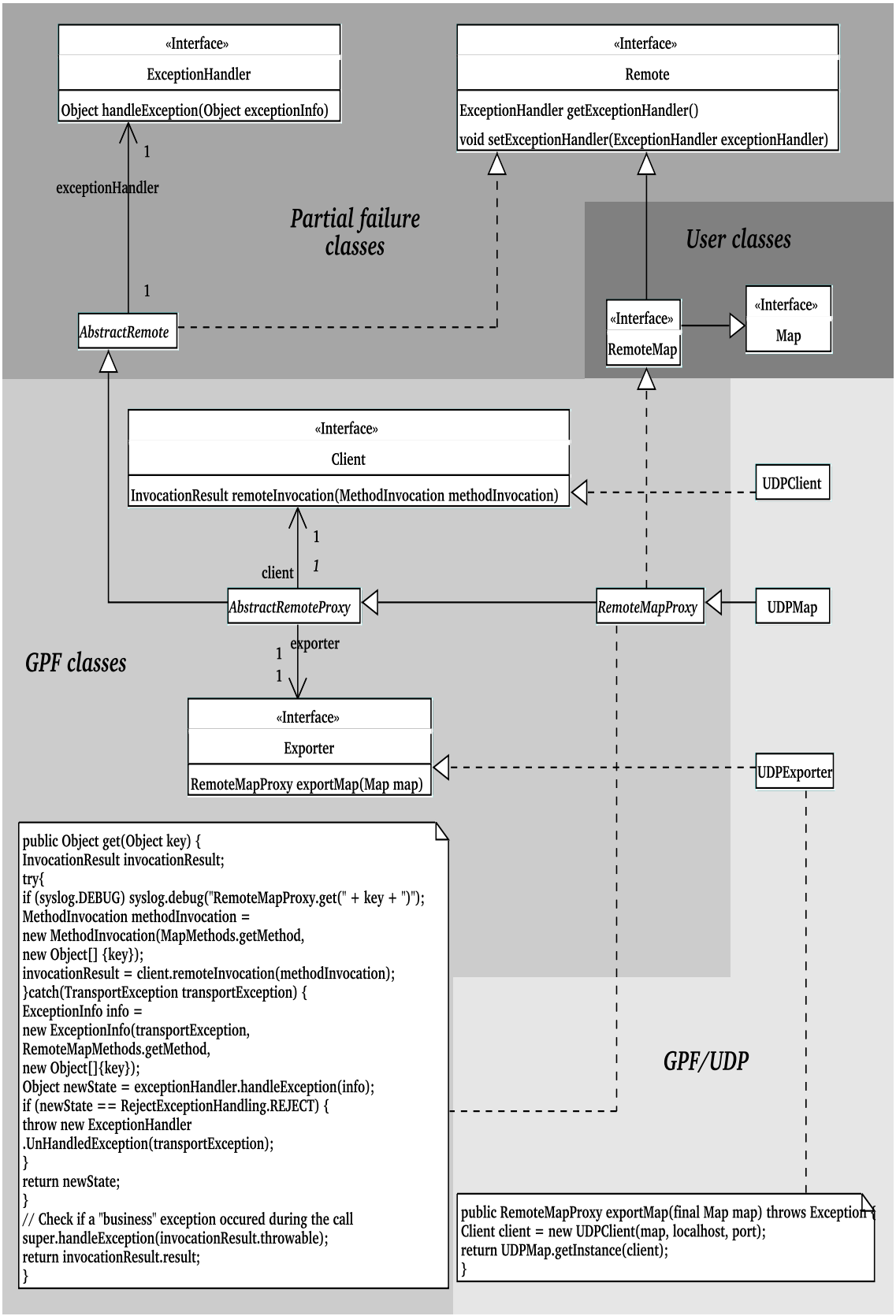


FIG. 10.2 – Architecture de classes pour l'implémentation UDP de `RemoteMap` utilisant le *Generic Protocol Framework*

l'ordre ou l'arrivée des messages envoyés par exemple. A ce titre, UDP est tout à fait adapté à un mode de communication asynchrone dans laquelle une panne partielle ne doit pas être considérée (lorsqu'un un réseau est supposé fiable, une erreur de communication est aussi grave qu'une corruption de la mémoire RAM d'une machine).

10.3.3 Détails sur l'implémentation UDP

Lorsqu'un objet *client* cherche à communiquer avec un objet distant *serveur*, deux étapes sont nécessaires :

1. la mise en place de la communication ;
2. la communication proprement dite.

La première étape est celle qui alloue les objets qui seront responsables de l'acheminement des messages et de la réception d'une réponse s'il y a lieu. Cela peut être la découverte d'un objet distant dans un service de nommage ou la réception d'une référence distante.

La deuxième étape se décompose en l'envoi de messages pour les appels de méthodes successifs et en la réception de messages pour la récupération d'éventuels résultats.

Dans les protocoles basés sur TCP (tel que RMI), il y a un nombre important de messages qui sont transférés sur le réseau pour assurer une communication fiable : les accusés de réception. Or, lors d'un appel de méthode, on peut considérer que le retour de l'appel constitue en lui-même un accusé de réception. C'est cette analyse qui est à l'orgine d'une implémentation de RMI sur UDP [117] dans laquelle les accusés de réception sont remplacés par les retours des appels de méthodes. Toutefois, ces travaux ont montré leurs limites dans un cadre général : il assez difficile d'assurer la fiabilité de la communication sous UDP sans utiliser des mécanismes de *timeout*, retransmission d'un message émis et annulation d'un message reçu.

De notre point de vue, ces problèmes ne sont pas liés au protocole UDP lui-même mais à l'utilisation d'UDP pour implémenter RMI. RMI est un protocole de communication synchrone, UDP, par contre, est intrinsèquement asynchrone. Tenter d'implémenter RMI au dessus du protocole UDP revient à implémenter toute une partie du protocle TCP qui assure la fiabilité du réseau. Par contre, nous allons utiliser les caractéristiques communes qu'ont JACOb et UDP : l'asynchronisme.

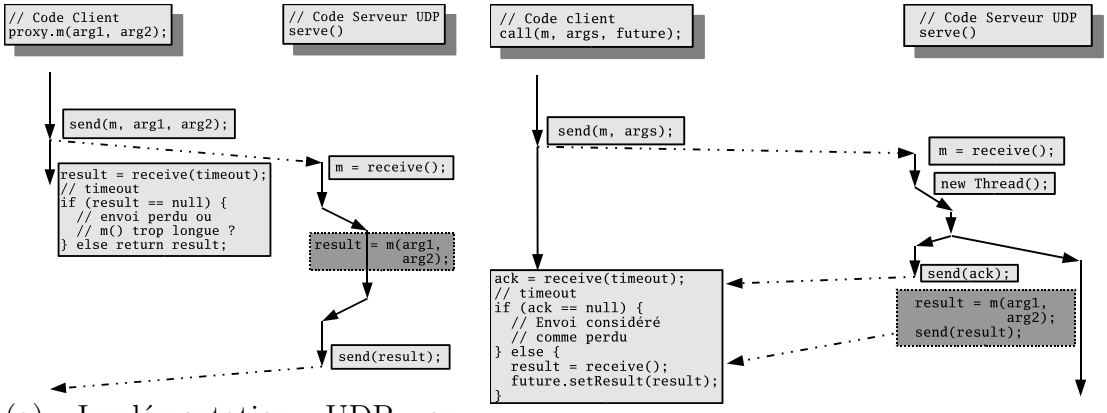
En reprenant l'idée originale, nous allons utiliser les retours des appels de méthodes comme accusés de réception. Toutefois, dans le cas synchrone, le nombre d'instructions entre le début de l'appel de la méthode et l'instruction `return` n'est pas borné[14] puisqu'il dépend de la méthode appelée. Aussi, lorsqu'un appel de méthode

[14]Mais peut être connu statiquement.

est effectué, le temps d'attente du retour de la méthode (donc de l'accusé) est difficile à évaluer. Dans le cas asynchrone, le nombre d'instructions est borné puisqu'il correspond à l'allocation de la thread d'appel[15] comme l'illustre le schéma §10.3 p110.

On voit apparaître sur ce schéma l'envoi d'un accusé de réception du serveur vers le client par l'instruction `send(ack)`. Cette caractéristique est très importante et absolument nécessaire. Elle permet au client de savoir que son appel a bien été pris en compte. Nous verrons que cet accusé est même fondamental puisqu'il permettra au client d'assurer la cohérence des objets envoyés en paramètres. Cet accusé sera également à l'origine d'un problème singulier trouvé dans la plupart des implémentations d'appel de méthodes distants asynchrones : l'asynchronisme *partiel* que nous présenterons en §12.1 p123 et que nous mettrons clairement en évidence en §19.2.2 p221.

En outre, l'accusé renvoyé aura une utilité puisqu'il permettra *l'annulation* d'appel asynchrone, mécanisme que nous décrirons en §14.5 p153.



(a) Implémentation UDP en mode synchrone

(b) Implémentation UDP en mode asynchrone

FIG. 10.3 – L'implémentation UDP est plus adaptée à un mode de communication asynchrone.

Finalement, un certain nombre de mesures de performances seront effectuées en §19 p211.

[15] Pas exactement comme nous le verrons en §15.2.3.2 p170.

10.4 Implémentation des références sur les stored objects

Nous avons vu que le concept de conteneur actif peut être utilisé comme un modèle mémoire d'une machine distribuée dans laquelle seuls des *stored objects* sont manipulés.

Dans cette optique, nous proposons la classe `StoredObjectReference` qui représente – comme son nom l'indique – une référence sur un *stored object*, c'est-à-dire un couple :

$$(activeMap, key)$$

La clé peut être choisie par l'utilisateur, mais peut aussi être créée par le système. Cette classe évite simplement d'avoir à préciser la clé à chaque manipulation d'un *stored object* et évite aussi la création de l'objet *futur* utilisé dans la méthode `call()`. Son code est présenté en Prog. 10.5 p111.

```
1  public class StoredObjectReference {
2      protected Object key;
3      protected ActiveMap activeMap;
4  ...
5      public Future call(java.lang.reflect.Method method,
6                         Object[] args) {
7          Future future = new Future();
8          activeMap.call(key, method, args, future);
9          return future;
10     }
11     public Object put(Object object){
12         return activeMap.put(key, object);
13     }
14     public Object get(){
15         return activeMap.get(key);
16     }
17     public Object remove(){
18         return activeMap.remove(key);
19     }
20     public boolean isValid() {
21         return activeMap.containsKey(key);
22     }
```

Prog. 10.5 – La classe `StoredObjectReference` (non définitive)

Nous verrons toutefois que cette classe n'est pas une simple classe utilitaire puisqu'elle implémentera l'interface `AsynchronousReference` que nous présenterons

en §14 p147 permettant de considérer ses instances comme des *références asynchrones*.

10.5 Caractéristiques de notre implémentation

Dans cette section les deux caractéristiques fondamentales de notre implémentation seront présentées : le *dynamisme* et le *partage des ressources*.

10.5.1 Dynamisme

JACOb possède une caractéristique que nous pensons être essentielle pour le développement d'applications distribuées orientées objets : le dynamisme. Dans JACOb, n'importe quel objet de l'utilisateur peut devenir accessible à distance dynamiquement sans que cet objet l'ait été prévu au départ. L'aspect distant n'étant plus nécessairement attaché à l'objet, il est possible de rendre distant un objet retourné par une méthode d'une classe de bibliothèque. Cette instance pourra être manipulée de manière distante et asynchrone ce qui offre un certain nombre d'avantages :

- effectuer du placement statique – c'est à dire à l'instanciation des objets – ou dynamique par le déplacement ou la réplication des objets d'un conteneur distant à un autre[16] ;
- augmenter le degré de parallélisme d'une application en utilisant des appels de méthodes asynchrones sur tout type d'objets[17] ;

Enfin, le développement d'applications distribuées orientées objets est facilité par la claire séparation entre la couche de gestion de l'aspect distant et la couche métier de la classe à développer.

Dans les travaux que nous avons étudiés, l'apport du *dynamisme*, est combiné :

- soit avec un certain nombre de limitations (plus d'exceptions dans CentiJ, limitations dans ProActive, passivité des conteneurs de JavaSpace, synchronisme des appels de méthodes dans Voyager) ;
- soit avec une plus grande complexité conceptuelle (critère d'activabilité de ProActive) ou logicielle (syntaxe des appels dans HORB ou dans RRMI).

Le *dynamisme* fournit par JACOb ne possède pas une grande complexité conceptuelle : la notion de conteneurs fait partie des connaissances de base de tout développeur. L'ajout de l'activité par l'intermédiaire de la méthode `call()` ne rajoute pas de difficultés particulières. L'aspect réflexif peut sembler déroutant pour un débutant, mais nous verrons en §16 p179 que nous le ferons disparaître au profit d'une syntaxe

[16] En utilisant l'un des paradigmes utilisant la mobilité de code et étudié en §3.3 p32.

[17] Les mécanismes de gestion de la concurrence que nous verrons en §IV p123 seront alors d'une grande utilité.

allégée assurant un typage fort. Enfin, les exemples de code présentés en §2.2 p8 et en §20 p227 montre que cette caractéristique n'augmente pas démesurément le code de l'utilisateur.

10.5.2 Ressources partagées

Dans le cadre du développement d'une application fortement distribuée dans laquelle un grand nombre d'objets sont amenés à devenir distants dynamiquement, l'utilisation du conteneur permet de partager de manière intrinsèque les ressources nécessaires à l'exportation de chaque objet.

Lors du développement d'une application, les problèmes liés à l'utilisation des ressources sont souvent gérés en aval, après un audit poussé de l'application en suivant le vieil adage :

> "dans une application, 80 % du temps exécute 20 % du code."

Aussi, l'optimisation de l'application ne doit être effectuée qu'après analyse de celle-ci, dans les fameux 20 % du code qui posent problème. Ces optimisations ne doivent pas remettre en cause l'architecture logicielle choisie au départ. Dans un cadre distribué, outre les problèmes algorithmiques habituels, l'exportation des objets peut amener des contraintes nouvelles si les ressources qu'elle nécessite ne sont pas partagées. Par exemple, dans le pire des cas, une telle exportation demande une nouvelle partie serveur (*cf.* §9.1 p79), impliquant la création d'un certain nombre de sockets, de descripteurs de fichiers et de threads pour la gestion de la concurrence (*cf.* §10.2.4 p104). Dans ce cas, l'exportation de nombreux objets engendre la création d'un grand nombre de ressources (mémoire ou système), qui peut facilement atteindre les limites de la machine virtuelle, du système ou du matériel.

Le partage de ressources fournies par l'utilisation des conteneurs actifs est de ce point de vue très efficace comme nous le verrons en §19.2.3 p223 dans la partie consacrée aux mesures de performance.

10.6 Services

Un certain nombre de services peuvent être proposés dans notre plate-forme. Par *service*, nous entendons des fonctionnalités utilisables de l'extérieur d'un composant logiciel. Un projet de fin d'étude d'ingénieur a été donné pour étudier une solution d'ajout de services dans la plateforme. Ce travail a permis de distinguer trois catégories de services :

Services associés aux méthodes Ces services sont utilisés systématiquement avant ou après un appel de méthode d'un conteneur. Par exemple, un service de

type *monitoring* peut être utilisé avant et après chaque appel à la méthode `call()` d'un conteneur actif pour un objet donné, afin d'étudier les performances d'une méthode particulière de cet objet.

Services événementiels Un certain nombre de services n'ont d'intérêt que s'ils sont liés à un événement particulier. C'est le cas par exemple d'un service de sauvegarde qui enregistre régulièrement le contenu d'un conteneur sur support non volatile. Un événement est donc généré à intervalle régulier afin d'avertir le service de sauvegarde.

Services génériques Les services de type générique ne sont associés ni à une méthode spécifique, ni à un événement paticulier. Ce sont des *stored objects* complètement quelconques qui offrent un certain nombre de fonctionnalités. Par exemple, un service de type instanciation à distance peut être implémenté comme un *stored object* contenant une méthode `instanciate()` prenant en paramètre un constructeur et les arguments à lui transmettre. Cet appel a pour effet d'instancier l'objet et de l'insérer dans le conteneur actif local au service. La clé est alors choisie par le service qui peut garantir son unicité puisqu'il a accès au conteneur localement : il peut donc placer un verrou et vérifier qu'une clé candidate n'est pas actuellement associée dans le conteneur. Ce service est d'une grande importance : il permet à tout objet – y compris à ceux dont la classe n'implémente pas l'interface `java.io.Serializable` – d'être accessible à distance. Jusqu'à présent, pour qu'un objet soit accessible à distance en utilisant le modèle des conteneurs actifs, il devait être inséré dans un conteneur actif distant en utilisant la méthode `put()` ce qui n'est possible que si l'objet est sérialisable. Nous proposons un tel service par le biais de la classe `Instanciator` brièvement présentée en §2.2.4 p15. En outre, ce service offre un gain non-négligeable en terme de performance lors du déploiement des objets : d'une part, il n'y a plus d'instanciation locale (puis insertion par copie), mais surtout, les instanciations peuvent être faites de manière asynchrone et donc potentiellement en parallèle. Nous en verrons une illustration en §20.3.3.1 p246.

On peut aussi distinguer trois classes de services :

Services indépendants Les services de ce type sont généraux et ne sont liés à aucune entité en particulier. Les services de nommage et de messagerie sont des exemples de services indépendants. Ils peuvent être fournis par des implémentations qui n'utilisent pas le concept de conteneur actif.

Service de conteneurs actifs Certains services n'existent que s'ils sont rattachés à un conteneur actif. Par exemple, un service d'administration du conteneur ou de

monitoring de son contenu dépend du conteneur particulier.

Services associés aux *stored objects* Enfin, un certain nombre de services sont associés à des *stored objects*. Un service de persistance qui enregistre sur une mémoire non-volatile l'état d'un objet stocké dans un conteneur en est un exemple.

Nous ne détaillerons pas l'implémentation de ces travaux, mais nous présenterons un certain nombre de services qui nous semble être importants et une implémentation possible.

10.6.1 Nommage

L'un des premiers services qui peut venir à l'esprit est le service de nommage qui permet d'enregistrer, de découvrir et de rechercher un objet sur le réseau (quel que soit l'objet : conteneur distant, *stored object*, référence RMI, etc.). De tels services sont déjà standardisés : par exemple, RMI propose `rmiregistry` et CORBA, le COS Naming Service. Enfin, Java fournit une couche interopérable supplémentaire avec l'API Java Naming Directory Interface qui adapte l'interface d'un service de nommage quelconque en une API standardisée dans le JDK. Par exemple, les services de nommage `rmiregistry`, COS Naming Service et LDAP sont tous employés avec les même instructions Java du point de vue de l'utilisateur du service.

JACOb utilise donc JNDI pour enregistrer un certain nombre d'objets primitifs distants. Les clients utilisent eux-aussi JNDI pour récupérer une référence sur un tel objet.

Notons qu'un conteneur distant de type `RemoteMap` peut faire office de service de nommage simplifié dans lequel la méthode `put` joue le rôle de `bind()`, et `get()` celui de `lookup()`. Le pattern *adapter* pourrait d'ailleurs être utilisé pour adapter un *proxy* de type `RemoteMap` en une classe dont l'interface serait conforme à une API déterminée (l'interface `javax.naming.Context` de l'API JNDI par exemple).

Une autre implémentation peut faire intervenir un *stored object* particulier possédant une référence locale sur un conteneur actif distant et qui joue le rôle du service de nommage. L'avantage d'une telle implémentation est de permettre l'utilisation d'appels de méthodes asynchrones facilitant ainsi le développement de solutions performantes de recherches ou d'enregistrements multiples.

10.6.2 Gestion de la sécurité

JACOb ne fournit pas de mécanisme de sécurité (autre que l'utilisation d'un `SecurityManager`). Ce problème est relativement important puisque la présence d'un conteneur actif distant facilite le développement et la propagation de *virus* :

il est en effet très facile d'insérer un objet dans un conteneur, d'utiliser un service de nommage pour découvrir un autre conteneur et de s'y propager. Un service de sécurité semble donc absolument nécessaire pour un déploiement de JACOb à grande échelle.

10.6.2.1 Accès au conteneur

L'administrateur d'un conteneur – *i.e.* celui qui l'instancie – peut vouloir autoriser l'accès à ce conteneur en fonction d'un certain nombre de paramètres qui peuvent être :

- la machine source de la requête ;
- l'utilisateur qui effectue la requête ;
- le type de la requête.

Ainsi, il pourrait être permis de consulter l'état du conteneur (utilisation des méthodes sans effet de bord), mais pas de le modifier directement sans l'utilisation d'un privilège particulier.

A ce titre, l'utilisation d'une classe englobante paramétrée par un certain nombre de règles de sécurité peut garantir un accès sécurisé au conteneur. Des mécanismes d'authentification peuvent être employés pour s'assurer de l'identité d'un client et lui appliquer les règles de sécurité qui le concernent. L'ensemble des opérations de sécurité peuvent être intégrées soit dans le protocole de communication bas-niveau lui-même (utilisation du GPF – vu en §10.3.2 p107 – pour y intégrer les sockets securisées SSL par exemple), soit dans l'API du conteneur englobant. Dans ce dernier cas, on peut imaginer une méthode de la classe englobante déclarée ainsi : l'objet

```
SecureSession connect(Login login, String passwd);
```

`SecureSession`, s'il est valide, (non-`null` par exemple) doit être passé en paramètre à chaque appel de méthode fonctionnelle (`put()`, `remove()`, etc.).

10.6.2.2 Accès aux objets

Lorsqu'un utilisateur a inséré un objet dans un conteneur actif, il ne souhaite peut-être pas en donner l'accés à tous les clients potentiels du conteneur. Il faut donc trouver un mécanisme de sécurité qui permette au créateur d'un *stored object* de définir un certain nombre de règles à appliquer aux utilisateurs de son objet. Pour cela, une spécialisation du conteneur s'impose afin d'appliquer aux clients de chaque appel de méthode (`put()`, `get()`, `remove()` mais aussi `call()` qui donne accès aux méthodes publiques du *stored object*) les bonnes règles. Par exemple, l'instanciateur de l'objet peut avoir tous les droits, y compris d'enlever (`remove()`) ou de remplacer (`put()`) l'objet, tandis que cette action est absolument interdite

à tous les autres clients. De même, les méthodes à effet de bord du *stored object* peuvent être accessibles aux clients appartenant à un groupe logiciel (groupe de type UNIX par exemple), mais interdite à tous les autres.

Cette fontionnalité peut être implémentée en utilisant les clés utilisées pour référencer les *stored objects*. Seules les méthodes `equals()` et `hashCode()` sont utilisées pour retrouver un *stored object*. Aussi, il est tout à fait possible d'utiliser deux types de clés, par exemple `RootKey` et `UserKey`, dont les méthodes `hashCode()` et `equals()` retournent les mêmes valeurs. Le type de clé permet alors de connaître les droits associés au client.

Ce dispositif n'augmente pas la complexité de l'usage du *stored object* puisque l'utilisateur final emploie de toute façon les mécanismes de transparence que nous verrons en §16 p179.

Chapitre 11

Conclusion

Dans cette partie, nous avons présenté une nouvelle abstraction : le *conteneur actif*. Si ce concept est directement issu d'une étude de la mobilité (*cf.* §II p27) et d'une modélisation du paradigme "agents mobiles" en π-calcul (*cf.* §C p295), il est généralisable aux appels de méthodes distants. Il possède une caractéristique fondamentale pour le développement d'applications distribuées : *le dynamisme*, cette faculté de pouvoir rendre distant n'importe quel objet. Cette caractéristique peut être exploitée sous de multiples formes : modèle mémoire distribué, accès distant multi-protocole dynamique, déploiement d'applications ou encore extension d'applications. En outre, en supportant directement les appels de méthodes asynchrones, les implémentations de ce concept offrent une plate-forme d'exécution parallèle. C'est dans ce cadre qu'un projet étudiant de maîtrise a été lancé pour expérimenter la plateforme dans une application de résolution du problème du voyageur de commerce. Ces travaux nous ont confronté à un problème de concurrence que nous allons examiner dans la prochaine partie.

Quatrième partie

Concurrence

Chapitre 12

Présentation du problème

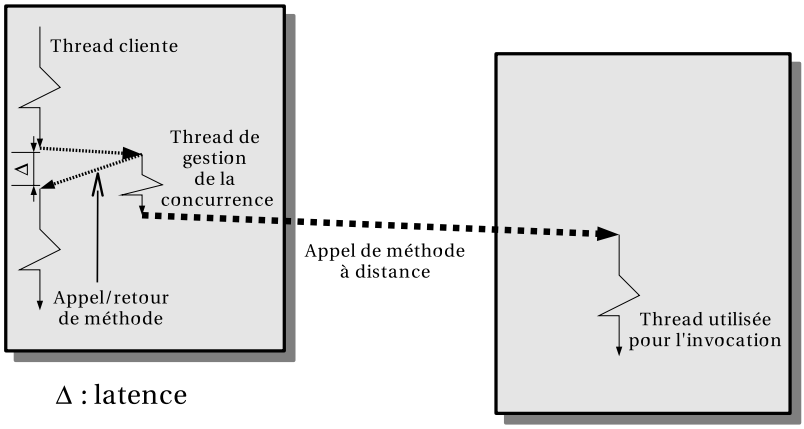
La partie précédente a présenté le concept de conteneur actif et son implémentation en Java, JACOb, qui trouve des applications dans le développement d'applications distribuées. Nous avons vu que la concurrence d'accès est une caractéristique fondamentale d'un système distribué (*cf.* §10.2.4 p104). Cette partie est dédiée à cet aspect.

12.1 Le problème rencontré dans JACOb

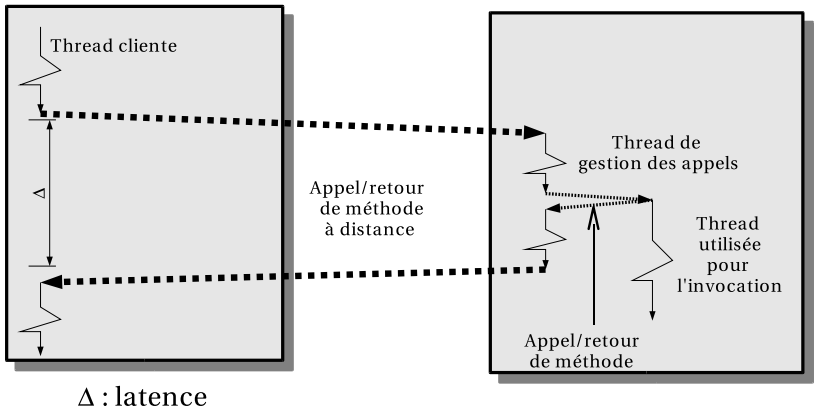
Comme mentionné en §9.11 p90, la méthode `call()` d'un conteneur actif peut être utilisée pour exploiter la concurrence d'une application. Si dans le cadre local, l'utilisation de cette caractéristique s'apparente au développement d'applications *multithreadées*, dans le cas où le conteneur est distant, même si l'appel de la méthode spécifiée dans `call()` est exécuté de manière asynchrone, le client reste bloqué, le temps que l'appel à `call()` soit initié côté serveur. Ce dernier étant distant, sa durée ne peut être négligée. On distingue donc deux types d'asynchronisme [176] illustrés par la figure Fig. 12.1 p124 :

- **Asynchronisme côté serveur :** la gestion de la concurrence est réalisée au niveau du serveur ce qui introduit une latence pour le client qui reste bloqué durant la transmission de l'appel.
- **Asynchronisme côté client :** la gestion de la concurrence est réalisée au niveau du client qui n'est pas bloqué, ni durant l'appel de méthode, ni durant la transmission de l'appel ;

Lorsque les deux types d'asynchronisme sont utilisés simultanément, nous parlerons d'asynchronisme *complet*. Nous verrons deux illustrations de ce mécanisme en §13.2 p142 et en §15.3 p172.



(a) Asynchronisme côté client



(b) Asynchronisme côté serveur

FIG. 12.1 – Les deux types d'asynchronisme dans le cas distant.

On pourrait vouloir privilégier l'asynchronisme côté client pour des raisons de performance : l'utilisation d'un mécanisme d'appel de méthode asynchrone permet de bénéficier de la concurrence qu'il engendre, et donc du parallélisme sous-jacent dans le cas distant. La latence réseau introduite par la communication dans le cas d'un mécanisme côté serveur uniquement peut limiter grandement l'intérêt même de l'appel asynchrone dans le cas où elle est très importante (lien de faible efficacité, protocole lourd ou machine destination surchargée par exemple). Cette latence a été mise en évidence lors de l'utilisation de JACOb dans un projet d'implémentation d'un algorithme *branch and bound* [207] pour la résolution du problème du voyageur de commerce [153].

Si la gestion de la concurrence côté serveur induit une latence qui pose de réels problèmes pour un certain type d'applications, elle reste néanmoins indispensable. En effet, elle permet de garantir au client qui le souhaite, que son appel a bien été

pris en compte par le serveur. Cette garantie fait souvent partie d'un certain nombre de modèles de programmation à base d'appel de méthodes asynchrones (ProActive par exemple). Aussi, il est important d'apporter les deux types d'asynchronisme. L'asynchronisme côté serveur est apporté par l'implémentation de JACOb. Nous devons donc nous pencher sur l'expression de la concurrence au niveau du client.

12.2 Programmation concurrente

La programmation séquentielle cède peu à peu la place à la programmation d'applications qui contiennent sous une forme ou une autre de la concurrence. La moindre interface graphique permet à la fois d'exécuter du code fonctionnel (télécharger une page web par exemple), et de répondre aux événements extérieurs (annulation par l'utilisateur par exemple). Dans le modèle orienté objets, deux vues complémentaires peuvent être clairement définies [123]:

- **vue centrée sur les objets (*object-centric*) :** le système est une collection structurée d'objets interconnectés ;
- **vue centrée sur les activités (*activity-centric*) :** le système est une collection d'activités potentiellement concurrentes.

Aucune des deux vues n'offre une vision globale du système, un objet pouvant être traversé par plusieurs activités, et une activité pouvant traverser plusieurs objets. Toutefois, chacune mène à deux enjeux distincts liés à la *correction* de programme :

- **l'innocuité (*safety*) :** qui évite la corruption des objets ;
- **la vitalité (*liveness*) :** qui évite l'inaction au sein des activités.

Des problèmes d'innocuité entraînent des comportements imprévus à l'exécution souvent liés à des données corrompues. Des problèmes de vitalité entraînent l'absence de comportement due à des étreintes fatales (*deadlocks*) : plus rien ne se passe ! Malheureusement, les solutions les plus simples pour améliorer l'innocuité peuvent avoir un impact négatif sur la vitalité et *vice-versa*.

Enfin, deux autres caractéristiques sont liées à la *qualité* du programme :

- **la réutilisabilité :** qui définit l'utilité d'un objet dans différents contextes ;
- **la performance :** qui définit la vitesse avec laquelle les activités s'exécutent.

Là encore, il est relativement compliqué, car généralement contradictoire, de vouloir simultanément augmenter ces deux critères de qualité.

Ainsi, le développement d'applications concurrentes est nettement plus complexe que celui d'applications séquentielles puisqu'il faut en permanence effectuer des compromis entre ces quatre enjeux : l'innocuité, la vitalité, la réutilisabilité et la performance. C'est probablement ce qui explique les adaptations pédagogiques nécessaires au contournement des difficultés spécifiques que rencontrent les étudiants dans ce domaine [29].

Un certain nombre de solutions à ces problèmes ont été proposées [124]. Elles mettent en œuvre des mécanismes d'exécution (threads, processus), des structures de données (verrous, sémaphores, ...), des *patterns* (*before/after pattern*, *fork/join pattern*, *atomic commitment pattern*, ...) ou des modèles (événementiel, processus, ...),. Nous allons nous focaliser sur les modèles de programmation qui permettent d'exprimer la concurrence dans une application (distribuée ou non).

12.3 Modèle de programmation

Si la classe `Thread` et le mot-clé `synchronized` du langage Java sont suffisants pour modéliser n'importe quel comportement concurrent, ils sont inadaptés à la programmation orientée objets et ont maintes fois fait l'objet de critiques [99, 108]. Aussi, plusieurs modèles ont été proposés pour améliorer l'expression de la concurrence, depuis fort longtemps. En 1978, Lauer et Needham [119] ont étudié les systèmes orientés *messages* et ceux orientés *procédures* afin de déterminer leurs différences. Aujourd'hui, on identifie les systèmes du premier type comme des systèmes basés sur un *modèle événementiel* et ceux du second type comme des systèmes basés sur un *modèle à base de processus*. L'étude a montré que les deux modèles sont *duaux*, à la fois en terme d'architecture et de performance. Malgré tout, le débat reste entier puisque l'on trouve toujours des partisans du modèle événementiel (passage de messages) [164] et du modèle à base de processus légers [28].

12.3.1 Modèle événementiel

La notion d'événement est assez intuitive dans les systèmes graphiques (événements souris, événements clavier) ou les systèmes qui gèrent des entrées/sorties (disques ou réseau par exemple)[1]. Par conséquent, le modèle événementiel a tendance à s'imposer de lui-même :

- il n'y a qu'un seul flot d'exécution et donc pas de concurrence CPU ;
- un certain nombre de gestionnaires d'événements sont enregistrés (*callbacks*) ;

[1] On peut d'ailleurs considérer que ce sont toujours des entrées/sorties, y compris dans les systèmes graphiques.

- le flot d'exécution consiste en une boucle qui notifie les gestionnaires enregistrés lorsqu'un événement survient ;
- il n'y a pas de préemption des gestionnaires d'événements ;
- l'exécution d'un gestionnaire est généralement une opération de courte durée.

Les gestionnaires d'événements implémentent donc la partie fonctionnelle de l'application.

Nous allons donner un aperçu des avantages et inconvénients de ce modèle.

12.3.1.1 Avantages du modèle événementiel

Faible consommation de ressources Les systèmes à base d'événements sont relativement "légers" dans la mesure où très peu de ressources sont consommées. Un seul flot d'exécution est utilisé, et l'essentiel des ressources consommées est lié à la création des événements. Ainsi, seule la mémoire a une importance dans ce type de système[2].

Pas de concurrence Le modèle n'utilisant qu'un seul flot d'exécution, le développeur se retrouve dans le cadre bien connu du modèle de développement séquentiel. Il n'y a pas de concurrence, donc pas de préemption, pas de synchronisation et pas d'étreinte fatale (*deadlock*).

Débogage simplifié Les dépendances temporelles sont liées aux événements, et non à l'ordonnancement interne. Les problèmes à résoudre sont donc généralement plus simples : par exemple, si un bouton ne répond pas du tout, c'est que l'événement n'a pas été géré. Dans le modèle dual, l'état d'un objet peut avoir été corrompu ce qui est nettement plus complexe à corriger.

Performance Le modèle événementiel évite le coût de l'obtention des verrous pour l'accès aux structures de données. De même, il n'y a pas la surcharge des changements de contexte.

Portabilité Le modèle événementiel est plus portable que le modèle à base de processus qui dépend du système sous-jacent.

12.3.1.2 Inconvénients du modèle événementiel

Malgré tout, l'approche événementielle soulève un certain nombre de problèmes :

[2]Évidemment, le code fonctionnel de l'application peut être, lui, grand consommateur de ressources, mais dans ce cas, et selon la relation de dualité, il en sera de même dans le modèle à base de processus.

Gestionnaires d'événements lourds Un gestionnaire d'événements qui prend trop de temps à s'exécuter peut rendre l'application inutilisable en donnant l'illusion qu'elle ne répond plus. Une solution est d'utiliser un sous-processus pour traiter l'événement, mais cela impose deux contraintes :

- la solution n'est plus conforme au modèle (un seul flot d'exécution) ;
- des problèmes d'innocuité peuvent apparaître si le gestionnaire tente de maintenir un état partagé entre ses sous-processus.

Pas de parallélisme possible Puisqu'un seul flot d'exécution est utilisé, il n'est pas possible d'exploiter simplement plusieurs processeurs. Là encore, une solution est de sortir du cadre du modèle et d'utiliser plusieurs flots d'exécution. Dans ce cas, le système doit gérer les problèmes d'accès concurrents aux structures de données, ce qui fait perdre de l'intérêt au modèle événementiel par rapport au modèle à base de processus.

Consommation CPU Un système d'entrées/sorties à base d'événements doit utiliser un mécanisme de scrutation (*polling*) pour éviter d'être bloqué. Par exemple, une application distribuée à base d'événements peut utiliser une seule thread d'exécution pour examiner l'ensemble de ses *sockets* à la recherche de données entrantes. Si une des *sockets* possède de telles données, la thread utilise le gestionnaire associé à l'événement reçu qui s'occupe de répondre. Le problème de cette approche est la consommation CPU induite par la scrutation[3]. Pour résoudre ce problème, il faut prendre un certain nombre de décisions sur l'insertion de code d'endormissement et de réveil (`Object.[wait|notify]()`, ou même `Thread.[sleep|yield]()`) et faire des choix empiriques sur les délais afin d'obtenir un bon compromis entre la consommation CPU et la réactivité du système. Cela complexifie nettement le développement de l'application et nuit à la portabilité de son code.

12.3.2 Modèle à base de processus

Utiliser un processus pour exprimer la concurrence n'est pas une idée nouvelle. Des mécanismes permettant la communication entre processus distincts ont été employés. La communication par sockets (type UNIX, ou INET) est encore de nos jours très utilisée. Toutefois le partage de mémoire entre processus distincts s'est avéré nécessaire pour des raisons de performance essentiellement. Si les IPC System V ont répondu à ce besoin dans un premier temps, la lourdeur du mécanisme liée à la protection mémoire assurée par le noyau[4] n'a pas favorisé son adoption. Surtout,

[3]Qui s'apparente dans le modèle *dual* à base de processus au changement de contexte.

[4]Et le mode protégé du processeur sous-jacent évidemment.

la généralisation de l'utilisation du concept de processus léger (ou de poids moyen selon le système), pour lequel l'ensemble de la mémoire du processus englobant est partagé, a facilité le développement d'applications exploitant la concurrence. A ce titre, Java a joué un rôle important en facilitant l'accès à ce style de programmation grâce à la relative simplicité du langage et à l'intégration de la concurrence dans l'API (classe `Thread`, moniteurs), dans le langage (mot clé `synchronized` et `volatile`) et dans la spécification de la JVM (ordonnancement, modèle mémoire).

Dans le modèle à base de processus, on distingue :

- plusieurs flots d'exécutions indépendants ;
- un état global partagé ;
- un ordonnancement préemptif des processus[5] ;
- un certain nombre de primitives de synchronisation (mutex, sémaphore, variables conditionnelles, ...).

Nous ne parlerons dans la suite que du modèle à base de processus légers (ou de poids moyen). En effet, aujourd'hui, l'utilisation de plusieurs processus lourds pour exprimer la concurrence dans une application (locale évidemment) ne se justifie plus : ils n'ont aucun des avantages des processus légers tout en ayant l'ensemble de leurs inconvénients. En particulier, le faible nombre d'opérations nécessaire à la gestion d'un processus léger – création et ordonnancement surtout – est sans commune mesure avec celui nécessaire à la gestion d'un processus lourd.

12.3.2.1 Avantages du modèle à base de processus

Nous pouvons énumérer les avantages liés à l'utilisation d'un modèle à base de processus légers.

Expressivité La notion de processus facilite l'expression de la concurrence en intégrant dans un tout (fonction ou méthode selon le langage), le comportement concurrent que l'on souhaite obtenir.

Passage à l'échelle (*scalability*) Lorsque les processus sont ordonnancés sur plusieurs processeurs, les performances globales de l'application peuvent être améliorées.

Recouvrement des entrées/sorties par du calcul Les opérations d'entrées/-sorties sont généralement coûteuses – et parfois même blocantes. Dans ce cas, générer

[5]Bien que non requis pour exprimer la concurrence dans un modèle à base de processus, l'ordonnancement préemptif est celui qui est le plus utilisé en pratique, et de fait, le seul disponible aujourd'hui sur les plate-formes standards Java.

un processus (léger de préférence) permet de recouvrir le temps de l'opération d'entrée/sortie par du calcul.

12.3.2.2 Inconvénients

Complexité La multiprogrammation est réputée difficile en raison de l'accès concurrent aux structures de données du processus lourd englobant. La gestion des synchronisation à ces accès est une tâche complexe à réaliser qui amène assez facilement à des problèmes d'innocuité ou de vitalité (*cf.* §12.2 p125).

Par ailleurs, le développeur doit prendre garde aux bibliothèques qu'il utilise. Toutes ne sont pas prévues pour être employées dans un environnement multithreadé. De plus, le débogage est lui aussi très compliqué puisque les contraintes temporelles de l'application ne sont pas les mêmes dans la version en cours de débogage et dans la version normale impliquant des ordonnancement de processus différents : un bug détecté à l'exécution peut disparaître au débogage et même d'une exécution à l'autre.

Limitations Le nombre de threads est limité. Par contre, cette limite dépend du type de threads :

- les processus de poids léger n'étant pas connus du système, seule la mémoire disponible est une limite à leur quantité ;
- les processus de poids moyen étant ordonnancés par le système, leur nombre dépend non seulement de la mémoire disponible mais aussi du nombre maximum autorisés pour l'utilisateur ou pour le système tout entier.

Enfin, dans la plupart des bibliothèques de threads, une estimation de la taille de la pile d'une thread doit être fournie à sa création[6]. Un compromis doit donc être trouvé entre le risque de dépassement de pile et le gaspillage mémoire engendré par une estimation trop large. Il est non seulement relativement difficile de faire une telle estimation, mais de plus, même si elle s'avérait judicieuse, elle ne serait pas portable.

Portabilité La portabilité des bibliothèques de threads est un problème majeur dans l'utilisation d'un modèle à base de processus légers. Tout d'abord, les APIs ne sont pas toujours identiques (WIN32, OS/2, Mach C, POSIX, ...). Par ailleurs, on trouve finalement assez peu d'implémentations entièrement conformes au standard POSIX[7] qui est la référence dans le monde *NIX.

[6] A moins d'utiliser celle proposée par défaut.

[7] Ce qui tend à faire penser que le standard des threads POSIX est finalement trop général, et qu'une autre API est préférable. C'est exactement la démarche de Linus Torvalds qui propose l'appel système `clone()` dans le noyau Linux en remplacement de l'appel `pthread_create()`.

Enfin, Java, fort de son slogan "*write once, run anywhere*", est directement confronté à ce problème de portabilité. Jusqu'au JDK v1.1, une implémentation en mode utilisateur était fournie : les *green threads*. Depuis le JDK v1.3, seules les threads natives sont supportées, généralement des threads noyau (processus de poids moyen) ce qui pose un certain nombre de problèmes, en particulier en ce qui concerne la gestion des priorités. Notre équipe s'est longuement intéressée à ce problème de spécification des threads dans la machine virtuelle Java [209].

Performance Obtenir de bonnes performances en utilisant un modèle à base de threads est un réel challenge. En effet, la concurrence des accès aux structures de données impose des points de synchronisation qui limitent la concurrence, et donc les performances globales de l'application. Par ailleurs, la gestion d'un grand nombre de threads demande un surcroît de travail à l'ordonnanceur pour sélectionner la prochaine[8] à exécuter. Si longtemps, ce problème est resté un obstacle à l'utilisation du modèle à base de threads dans les applications de type serveurs haute performance, contredisant ainsi la propriété de dualité énoncée en §12.3 p126, il semble qu'il soit résolu par une implémentation efficace d'une bibliothèque de threads [28].

12.3.3 Particularité dans Java

De notre point de vue, la notion de thread pour exprimer la concurrence est inadaptée à la programmation orientée objets. La représentation d'une thread par un objet est relativement délicate, étant donnée la nature éphémère de l'entité *processus léger*. Lorsque l'entité a disparu du système, que doit-on faire de l'objet qui le représente ? Que deviennent les références à cet objet ?

En Java, l'entité est représentée par une instance de la classe `java.lang.Thread` laquelle contient la méthode `isAlive()` qui permet de connaître son état[9]. Toutefois, cette méthode est d'une utilité limitée puisque l'information récupérée peut être périmée à tout moment.

Enfin, au niveau syntaxique, il existe en Java un certain nombre d'incohérences qui perturbent les débutants :

- l'instruction `new Thread(...)` n'a pas pour effet de créer le processus léger sous-jacent, seulement l'objet qui le représente ;
- le code "métier" de la thread peut être dans une sous-classe de la classe `Thread` (héritage), ou dans une instance de classe qui implémente l'interface `Runnable` (délégation) ;

[8] ou les prochaines sur un système à plusieurs processeurs.

[9] Une thread est dans l'état *vivant* si elle a été démarrée et si elle n'est pas encore *morte*.

- la création du processus léger est réalisée à l'aide de la méthode `Thread.start()` qui est à usage unique, ce qui implique qu'une instance de la classe `Thread` ne sert pas de squelette à la création de processus légers.

Notons que si la création d'un processus léger est une opération peu coûteuse, en Java elle implique la création d'une instance de la classe `Thread` ou d'une sous-classe[10]. De même l'arrêt d'un processus léger entraîne l'éventuelle libération de l'objet qui le représente s'il n'est plus référencé. De ce point de vue, l'entité thread en Java n'est pas si "légère".

Par ailleurs, l'API a dû revenir sur un certain nombre de choix réalisés en fonction d'une correspondance avec les processus légers sous-jacents fournis par le système Solaris sur lequel les premières machines virtuelles Java ont été développées. C'est ainsi que l'on a retrouvé les méthodes `Thread.suspend()` et `Thread.resume()`, directement issues des fonctions `thr_suspend()` et `thr_resume()` de l'API des threads Solaris qui sont aujourd'hui dépréciées [142].

12.3.3.1 Swing, le mauvais exemple

On pourrait se demander pourquoi les développeurs des *Java Foundation Classes* – plus connues sous le nom de *Swing* – ont réalisé leur architecture selon un *modèle événementiel* (*cf.* §12.3.1 p126) – aussi appelé *single-threaded event model*. Java fournissant les threads, pourquoi ne pas les avoir utilisées ? Nous avons donné plusieurs arguments en faveur et en défaveur de chacun des deux modèles (*cf.* §12.3 p126). Manifestement, l'API pose des problèmes.

Par exemple, l'ensemble des opérations graphiques sont effectuées par la seule thread Swing : elle récupère les événements graphiques et notifie les gestionnaires d'événements. Il est donc évident que les gestionnaires d'événements lourds, demandant beaucoup de cycles processeurs pour être achevés, ne peuvent être exécutés par la thread Swing. Aussi, il est nécessaire de sortir du modèle théorique et de générer des threads pour la gestion de ces événements dits lourds. Il est donc tout aussi nécessaire de maintenir un minimum de synchronisation entre les threads de traitement lourd et la thread Swing ce qui complique le code qui est un mélange de programmation événementielle et de programmation à base de processus. Pire, certaine méthodes de l'API Swing sont synchrones[11] sans distinction syntaxique par rapport aux autres méthodes qui, pour la plupart, sont asynchrones : c'est le cas de la méthode `javax.swing.TextComponent.setText()` par exemple. Seule la documentation de l'API permet de connaître la sémantique (synchrone ou asynchrone)

[10] Si le code métier est dans une classe qui implémente l'interface `Runnable`, la création d'un processus léger peut entraîner la création de deux objets.

[11] Elles génèrent un événement graphique et attendent qu'il soit géré.

d'un appel de méthode[12].

Récemment Jonathan Simon [188] a donné une raison de la complexité de l'écriture des applications Swing : de son point de vue, ces applications sont écrites de manière séquentielle en utilisant un modèle concurrent à base d'événements. Il faudrait donc se conformer au modèle événementiel et utiliser si besoin une bibliothèque spécialisée comme *Java Messaging Service*[13]. Le problème est qu'un certain nombre d'applications (ou de fragments d'applications) s'écrivent difficilement selon un modèle événementiel, et ne sont pas vraiment concurrentes.

Pour simplifier le développement des applications graphiques à base de composants Swing, plusieurs solutions sont proposées. En premier lieu, Sun fournit la classe `SwingWorker` [145], une classe générique non livrée avec le JDK[14] qui simplifie le code des gestionnaires d'événements graphiques dit "lourds".

Surtout, plusieurs travaux tentent de rendre la programmation graphique dans Java complètement synchrone. C'est le cas par exemple de Foxtrot [1] qui connaît un succès considérable et de Spin [2] qui reprend le modèle synchrone de Foxtrot mais en améliore certain aspects (gestion transparente des exceptions notamment).

Aussi, contrairement à Joseph Bowbeer [36], nous pensons que les discussions autour du développement des applications graphiques utilisant Swing sont loin d'être terminées.

12.4 Modèle utilisant le passage de messages

L'appel de méthode entre objets est souvent identifié comme du *passage de messages*. Ce paradigme rentre donc dans la catégorie du modèle événementiel tel que décrit en §12.3.1 p126. Cependant, la version asynchrone de ce paradigme est considérée comme étant du second type – c'est-à-dire à base de processus.

L'invocation de méthode asynchrone offre un certain nombre d'atouts dans l'expression de la concurrence dans un langage orienté objets. En premier lieu, l'appel de méthode (synchrone) étant un modèle bien connu des développeurs, sa version asynchrone n'offre pas de grandes surprises puisqu'il en reprend la syntaxe et une partie de la sémantique en ce qui concerne le passage des paramètres (par valeur ou par référence). Par ailleurs, la généralisation du paradigme *appel de méthode à distance* tend à montrer que l'appel de méthode standard s'étend relativement bien

[12]Il en est d'ailleurs de même avec l'API de la classe `java.lang.Thread` dont la méthode `start()` est asynchrone contrairement à sa méthode `run()`. Cependant, étant donné l'entité représentée par les instances de cette classe, le problème est relativement mineur.

[13]JMS supporte les deux modèles de communications *publish/subscribe* et point à point, et permet l'usage d'objets Java quelconque pour typer les messages.

[14]Pourquoi ? Mystère ...

au cas distant. Il est donc tentant de penser qu'il en est de même avec la version asynchrone du paradigme.

Finalement, nous avons vu en §10.3.2 p107 que certains protocoles de communication sont très bien adaptés à un mode de communication asynchrone (UDP en l'occurrence), ce qui laisse entrevoir des optimisations possibles.

12.4.1 État de l'art

Nous allons présenter les travaux qui utilisent l'appel de méthode asynchrone comme paradigme de programmation, et que nous considérons être les plus significatifs.

12.4.1.1 Objets actifs

La notion d'objets actifs est apparue grâce aux travaux menés par Denis Caromel [42] dans Eiffel//. Une version en C++ a ensuite été proposée (C++//). Aujourd'hui, le concept d'objet actif est fourni dans la plate-forme ProActive [41] (anciennement Java//) entièrement écrite en Java. Nous avons déjà parlé de ProActive en §9.3 p82 à propos des appels de méthodes distants.

Un objet actif est un objet qui contient sa propre thread d'exécution. Les clients d'un objet actif ne peuvent invoquer directement les méthodes d'un objet actif : l'objet devient actif car c'est lui, par l'intermédiaire d'une thread particulière, qui invoque ses propres méthodes et non la thread qui fait l'appel[15]. Les clients invoquent une méthode de manière transparente et asynchrone. Le résultat de cet appel est un objet *futur* [222], qui utilise le mécanisme d'*attente par nécessité* : lorsqu'une méthode de l'objet *futur* est invoquée, l'appelant est bloqué jusqu'à ce que l'appel soit effectivement terminé. Ce mécanisme sera étudié plus en détail en §16.1 p180.

Un objet passif est un objet qui ne contient pas de thread.

Un objet actif est composé d'un corps (*body*) dont le rôle est d'enfiler les requêtes d'appels de méthodes et d'exécuter celles-ci dans un ordre défini par une politique d'ordonnancement (par défaut une politique de type FIFO). Les clients utilisent un *proxy* dont le rôle est de générer les objets *futur* renvoyés au client pour le résultat, transformer les requêtes en appels de méthodes (sous forme de méta-objets [109], par réification), et de copier en profondeur les arguments des appels de méthodes pour les transférer au corps. La figure FIG. 12.2 p135 illustre la différence entre un appel de méthode sur un objet passif et sur un objet actif.

[15] A ce titre, le terme *actif* aussi utilisé pour nos conteneurs n'a pas un sens si différent. Un conteneur actif est un objet actif particulier.

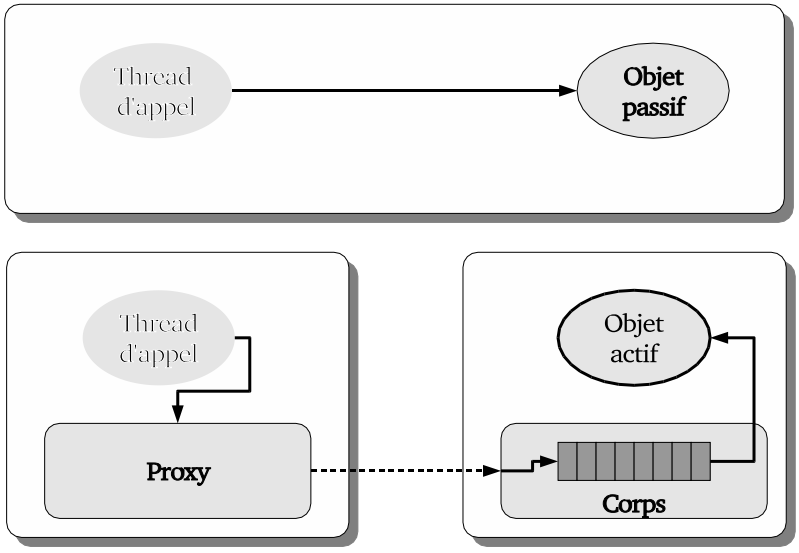


Fig. 12.2 – Différence entre un appel de méthode sur un objet passif et sur un objet actif.

Problèmes Nous avons évoqué un certain nombre de problèmes inhérents au modèle d'objets actifs en §9.3 p82.

Étreinte fatale : le premier problème vient du faible degré de concurrence que possède un objet actif puisqu'il ne peut posséder qu'une seule thread[16]. Outre le problème de performance, l'utilisation d'une seule thread pour l'invocation des méthodes d'un objet actif peut amener à des étreintes fatales comme dans le scénario suivant, illustré par la figure Fig. 12.3 p136. Dans ce scénario, deux objets actifs `a` et `b` se référencent mutuellement. La méthode `foo()` de `a` est invoquée de manière asynchrone. La méthode `foo()` est invoquée par la thread de l'objet actif, elle demande l'invocation asynchrone de la méthode `mb()` de l'objet `b`. L'objet `b` étant actif, cet appel n'est pas bloquant, et la thread appelante poursuit son exécution jusqu'à atteindre l'instruction `rb.f()`. Là, le mécanisme d'attente par nécessité est mis en œuvre : l'appel à `b.mb()` n'est pas terminé (et n'a probablement pas encore commencé) et la thread est bloquée en attente du résultat. Pendant ce temps, l'objet actif exécute la méthode `mb()`, qui à son tour invoque de manière asynchrone la méthode `ma()` sur l'objet actif `a`. Le même mécanisme aboutit au blocage de l'objet actif `b` sur l'instruction `ra.g()`. Le problème est que la méthode `ma()` ne sera jamais

[16]Notre équipe a récemment étendu le critère d'activabilité (que nous verrons en §12.4.1.1 p137) aux objets multithreadés. Cependant, dans la définition originale, le degré de concurrence d'un objet actif est nul.

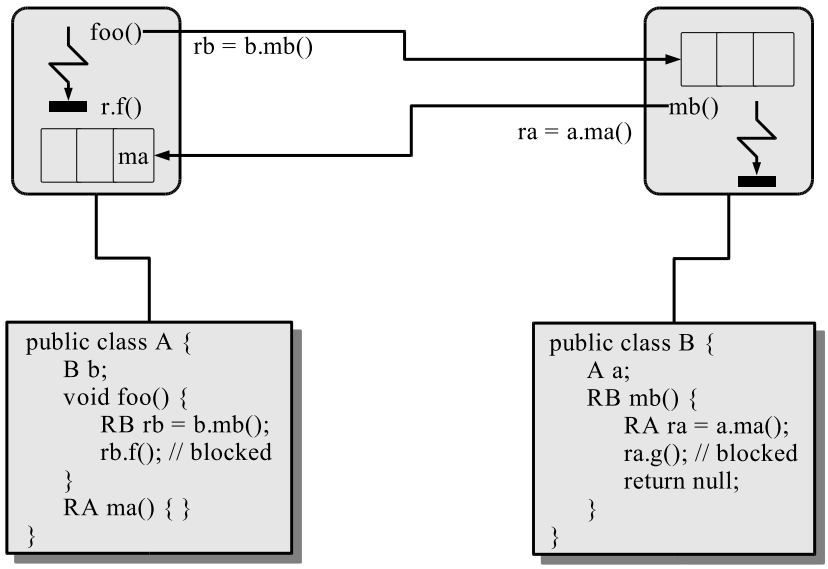


FIG. 12.3 – Un scénario possible menant à une étreinte fatale dans le modèle des objets actifs.

exécutée, la thread de l'objet actif `a` étant déjà bloquée sur le résultat de l'appel à `b.mb()`. D'où une étreinte fatale !

Gestion des exceptions : l'autre problème vient de la récupération des exceptions. Puisque l'invocation est asynchrone, la thread appelante n'est pas bloquée et peut donc sortir du bloc `try/catch` qui englobe l'appel, comme illustré par le code PROG. 12.1 p136.

```
1  int b = ...; // byte to write
2  java.io.FileOutputStream out = null;
3  try{
4      out = new java.io.FileOutputStream(...);
5      out.write(b);
6  }catch(java.io.IOException ioe) {
7      // handle any IO exception
8  }catch(java.lang.SecurityException se) {
9      // handle security exception
10 }
```

PROG. 12.1 – Problème de la gestion des exceptions dans le modèle des objets actifs

Une solution (retenue par ProActive) est de rendre synchrone toutes les méthodes susceptibles de lever une exception (donc, en Java, celles qui ont une clause `throws`

à leur déclaration). On perd ainsi la concurrence, mais on assure une sémantique correcte à l'exécution : l'appel étant synchrone, l'exécution se passe comme dans le cas séquentiel.

Cette approche soulève le problème n°12 p84 : une synchronisation explicite est nécessaire pour récupérer une exception non contrôlée. En Java toute méthode peut lever une exception non contrôlée. Le mécanisme standard de récupération des exceptions ne peut fonctionner lorsque l'exception levée par une méthode d'un objet actif est de type non contrôlé. La méthode étant asynchrone, il faut nécessairement utiliser une synchronisation explicite pour éviter de sortir du bloc `try/catch` englobant comme illustré par le code Prog. 12.2 p137. A moins de rendre toutes les méthodes synchrones, il n'y a pas, à ce jour, de solution raisonnable à ce problème.

```
1  java.util.List list = library.getAList();
2  ...
3  try{
4      // Asynchronous call
5      Object o = list.remove(0);
6
7      // *Must* wait the result (in case of a runtime exception)!
8      o.toString(); // May also use 'o.waitFor()' but a cast is needed!
9  }catch(UnsupportedOperationException e) {
10     // This exception is a runtime exception
11     list = library.getAList(); // For example!
12 }
```

Prog. 12.2 – La récupération d'une exception non contrôlée requiert une synchronisation explicite.

Critère d'activabilité Afin de faciliter le développement d'applications concurrentes et/ou réparties (et donc parallèles), l'équipe OASIS fournit une preuve qui établit qu'une application parallèle est similaire d'un certain point de vue à sa version séquentielle. Ils proposent un critère [22] qui permet de déterminer quand un objet peut être rendu actif sans modifier le comportement de l'application. Sommairement, un objet est *activable* lorsque son ensemble d'accessibilité est clos, c'est-à-dire, lorsque tous les objets qu'il utilise ne sont utilisés que par lui. La preuve consiste à montrer que le graphe du sous-système[17] considéré est un arbre. Il est important de noter que selon ce critère, les objets `a` et `b` du problème d'étreinte fatale illustré par la figure Fig. 12.3 p136 ne peuvent pas être actifs tous les deux

[17]L'ensemble des objets utilisés par un objet.

en même temps. En effet, puisque nous avons un cycle dans les deux graphes des sous-systèmes des deux objets, seul l'un d'entre eux est activable.

Le critère d'activabilité est restreint à un sous-ensemble du langage Java dans lequel il n'y a ni thread, ni méthode ou champ de classe, ni chargement dynamique de classe. Dans la pratique, il est relativement complexe de se conformer à ces limitations. Par exemple, comment savoir qu'un objet n'est pas – et ne sera jamais – parcouru par plusieurs threads simultanément ? Certaines instances de classes de l'API Java contiennent des threads sans que cela soit mentionné dans la documentation[18]. Il est donc relativement compliqué de développer sous ces contraintes sans un outil automatique capable, par analyse du code (statique ou dynamique) de vérifier qu'un objet est activable ou non. Un tel outil est en cours d'élaboration dans l'équipe d'OASIS [22]. Par ailleurs, les travaux de Pascal Grange dans notre équipe vont dans ce sens en étendant la propriété d'activabilité à un contexte multithreadé [47].

12.4.1.2 Acteurs

Le concept *acteurs* a été proposé par Carl Hewitt en 1977 [97]. Un modèle de calcul plus précis et formalisé a ensuite été défini par Gul Agha et Carl Hewitt en 1986 [16, 17]. Le terme *acteur* peut donc être employé dans un sens générique tel que défini à l'origine ou en référence à la sémantique précise du modèle de calcul des acteurs. Pour l'essentiel, un acteur est un objet actif qui communique exclusivement de manière asynchrone. Dans un tel système, il n'y a pas d'objet passif, tous les objets sont actifs. Le mode de communication entre acteurs utilise un système de boîtes aux lettres. Le système de courrier garantit que les messages seront acheminés vers leur destinataire, mais dans un ordre *indéterminé*. Les actions que peuvent réaliser un acteur sont limitées :

- envoyer un message à lui-même ou à un autre acteur ;
- créer de nouveaux acteurs ;
- spécifier le *comportement de remplacement*.

Un *comportement* traite un et un seul message. A un moment donné, un acteur a un comportement courant. Le comportement courant spécifie le comportement de remplacement, c'est à dire le comportement qui traitera le message suivant.

Le concept de comportement de remplacement unifie :

- **le changement d'état :** le comportement de remplacement peut être une copie du comportement courant ou manifester un changement quelconque ;
- **la concurrence intra-objet :** dès que le comportement suivant est spécifié, le traitement du message suivant peut débuter ;

[18] C'est par exemple le cas pour la classe `java.security.SecureRandom`

• **la synchronisation :** tant que le comportement de remplacement n'est pas spécifié, le message suivant ne peut être accepté.

Une extension au langage Java a été proposée [210]. D'une part, cette solution n'a jamais été implémenté, d'autre part, nous cherchons à nous limiter à une solution 100% Java.

12.4.1.3 Objets séparés

Le modèle à objets séparés a été introduit par Bertrand Meyer [137] comme une extension du langage Eiffel, permettant l'exécution systématiquement asynchrone des appels de procédures (méthodes déclarées retournant `void`). Ces travaux ont été repris pour être adaptés au langage Java [108].

Un objet séparé est une instance de classe séparée (elle implémente l'interface `Separate`). Dans une telle classe, toutes les méthodes de type *procédure* sont exécutées de manière asynchrone. Les autres méthodes sont exécutées de manière synchrone. Plusieurs appelants peuvent partager le même objet séparé. La sémantique des appels de procédure d'un objet séparé garantit que du point de vue de l'appelant, les appels seront exécutés dans l'ordre reçu par la cible. Toutefois, afin d'éviter les problèmes d'étreinte fatale dans les appels récursifs, la sémantique est modifiée de la façon suivante :

Un objet séparé qui effectue un premier appel à un autre objet séparé est appelé l'*origine* de la séquence des appels. Lorsqu'un objet séparé qui est en train d'exécuter une méthode reçoit une requête provenant de la même origine, alors il exécute immédiatement l'appel correspondant à la requête.

Par ailleurs, une nouvelle syntaxe est introduite dans le langage Java afin de prendre en compte les problèmes de synchronisation. Les points de synchronisation sont identifiés par des pré-conditions : des expressions Java dont l'évaluation retourne une valeur booléenne. L'exécution d'une méthode n'a lieu que si l'évaluation de la pré-condition qui lui est associée renvoie une valeur vraie. Enfin, une pré-condition peut avoir une sémantique transactionnelle lorsqu'elle implique un objet séparé :

si la pré-condition d'une méthode `f()` d'un objet séparé `s` inclut d'autres objets séparés `s1` et `s2` par exemple, alors durant l'exécution de `f()`, aucun appel d'un autre objet séparé que `s` ne sera exécutée par `s1` et `s2`.

Ainsi, plusieurs méthodes peuvent être invoquées sur un objet séparé en ayant la garantie qu'aucune autre méthode ne sera exécuté pendant leur exécution (d'où

l'aspect transactionnel). C'est en quelque sorte une *réservation* d'utilisation d'un objet.

La gestion des exceptions levées par une méthode procédurale invoquée de manière asynchrone n'est pas clairement définie. Cela semble lié à la grande différence qui existe entre les exceptions du langage Java et celles du langage Eiffel [111].

Problèmes Le problème majeur du modèle des objets séparés est qu'il est limité aux méthodes procédurales ce qui réduit son domaine d'applicabilité : sur les 17254 méthodes publiques que comportent le JDK v1.4[19], seules 34 % sont procédurales. Parmi celles-ci, un grand nombre n'ont aucun intérêt à être parallélisées :

`Object.wait()`, `Object.notify()`, `Object.finalize()`,
`Thread.start()`, `Thread.interrupt()`,
`*Stream.close()`, `*Stream.flush()`,
`*Reader/*Writer.close()`.
...

Pire, ce sont justement celles qui n'ont aucun intérêt à la parallélisation qui sont très fréquemment employées.

Enfin, la modification du langage Java ne permet pas la réutilisation de code dont on ne possède pas les sources. Or, nous avons vu que l'aspect *dynamique* est très important.

Par conséquent le modèle à objets séparés est intéressant dans la mesure où le code est écrit en fonction de ce modèle (méthodes procédurales en particulier).

12.4.1.4 Autres travaux

D'autres résultats sont présentés dans la littérature. Cependant, ceux que nous avons étudiés sont généralement dédiés à la communication distante. CORBA [218], par exemple, offre un service d'invocation de méthode asynchrone [182], mais il est voué à une utilisation distante. De même, ARMI (*Asynchronous Remote Method Invocation*) [65] propose une extension du langage Java à l'aide du mot clé `asynch` pour déclarer les méthodes des objets qui doivent être invoquées de manière asynchrone. Une réimplémentation des *stubs* et *skeletons* de RMI et l'utilisation d'objets *futurs* permettent aux développeurs d'utiliser le paradigme d'appels de méthodes asynchrones. Cette solution n'offre pas le *dynamisme* recherché (*cf.* §10.5.1 p112). En outre, elle est liée au cas distant. Cette remarque a également été faite pour RRMI [203] en §9.9 p89.

La liste des travaux qui utilisent le paradigme d'appel de méthodes asynchrones est longue et ne peut être énumérée de manière exhaustive.

[19] Seules les classes du JDK dont le nom contient le préfixe `java` ont été considérées.

Chapitre 13

Solutions possibles

Dans notre recherche d'une solution pour l'expression de la concurrence au sein de JACOb, plusieurs options se sont offertes à nous. Une description et une évaluation de ces options est proposée dans ce chapitre. L'évaluation tient compte d'un certain nombre de facteurs de génie logiciel (réutilisabilité, robustesse, performance, maintenabilité, ...), mais surtout de l'adéquation de la solution avec le concept des conteneurs. En outre, le *dynamisme* doit évidemment, être préservé.

13.1 Utilisation de tâches

La première solution retenue a été une implémentation *ad-hoc* à base de tâches. Une tâche représentait une opération à effectuer sur le conteneur de manière asynchrone. Nous avions donc défini une classe abstraite `Task` contenant le code relatif à la gestion de la concurrence : création de la thread, attente et scrutation du résultat, etc. Les classes fonctionnelles héritant de la classe `Task`, il y avait une classe par opération élémentaire : `GetTask`, `PutTask`, `RemoveTask` et `CallTask`. En effet, si le problème d'asynchronisme *partiel* existe dans l'appel asynchrone *côté serveur* `call()` (*cf.* §12.1 p123), la situation est complètement synchrone pour les autres opérations. Cette solution permettait donc d'invoquer toutes les méthodes du conteneur de manière asynchrone.

Ainsi, L'API était assez simple :

```
Task task = new PutTask(activeMap, key, myObject);
```

préparait l'association via la méthode `put()` de la clé `key` avec l'objet `myObject` dans le conteneur spécifié par le paramètre `activeMap`. L'instruction `task.start()` démarrait l'appel, la thread courante (côté client) n'étant pas bloquée. Un certain nombre de méthodes permettaient ensuite de scruter ou d'attendre le résultat de l'appel.

13.1.1 Avantages

Cette solution a l'avantage d'être rapidement implémentée. Son efficacité dépend essentiellement de l'architecture sous-jacente qui génère la concurrence. Il va sans dire qu'une implémentation à base de *threads pool* est préférable à une simple implémentation à base de threads.

13.1.2 Inconvénients

Cette solution possède l'inconvénient d'être peu générique. Par exemple, chaque nouvelle méthode ajoutée à la classe `ActiveMap` nécessite un nouveau type de tâche et donc une nouvelle sous-classe de `Task`. Par ailleurs, nous avons vu que l'ensemble du paquetage `Collection` est proposé en version distante dans JACOb. Il peut donc être intéressant de fournir des tâches pour chaque opération de chacune des collections.

13.1.3 Conclusion

La solution à base de tâches a vite montré ses limites. Nous nous sommes alors penchés vers une solution plus générique. Nous allons voir qu'en utilisant le concept de conteneur dans un contexte local, nous obtenons une solution intéressante.

13.2 Utilisation de conteneurs actifs locaux

Nous avons cherché à obtenir une solution générique : n'importe quel objet devrait pouvoir être accédé de manière asynchrone. C'est exactement la caractéristique d'un conteneur actif local : la méthode `call()` est asynchrone ! Par conséquent, pour qu'un objet local soit accessible de manière asynchrone, il suffit de l'insérer dans un conteneur actif local et d'utiliser la méthode `call()` pour les communications. L'utilisation d'un conteneur actif distant pose un problème d'asynchronisme *partiel*. Or, l'accès à un conteneur actif distant se fait par l'intermédiaire de son *proxy* (*cf.* §10.3 p104).

Pour obtenir un asynchronisme *côté client* des méthodes d'un conteneur distant, il suffit donc d'insérer dans un conteneur *local* le proxy d'un conteneur *distant*.

Il y a deux remarques à faire en ce qui concerne la méthode `call()` :

- il faut *chaîner* deux appels à `call()` pour contacter un *stored object* dans le conteneur distant ;

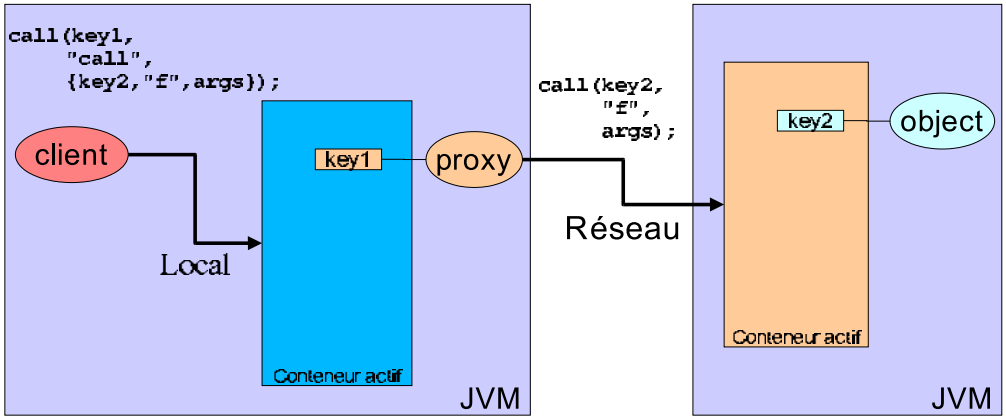


FIG. 13.1 – Asynchronisme *complet* avec des conteneurs actifs locaux.

- l'asynchronisme d'un tel appel est à la fois *côté client* par le conteneur actif local et *côté serveur* par le conteneur actif distant – c'est donc un asynchronisme *complet* (*cf.* §12.1 p123).

La figure FIG. 13.1 p143 illustre ce mécanisme.

13.2.1 Avantages

L'avantage de cette approche est de permettre à tout objet d'être accessible de manière asynchrone. Par conséquent, l'ensemble JACOb constitué des conteneurs actifs locaux et distants permet à tout objet d'être invoqué de manière distante et asynchrone. C'est l'aspect *dynamique* qui reste la caractéristique fondamentale de notre solution.

13.2.2 Inconvénients

Il y a plusieurs inconvénients à notre approche. Le premier est la lourdeur syntaxique du mécanisme. Pour invoquer la méthode `foo(new StringBuffer("Dummy"))` d'un *stored object*, il faut encapsuler l'appel à la méthode `call()` du conteneur distant dans l'appel à la méthode `call()` du conteneur local, ce qui s'écrit :

```
1  MyClass object = new MyClass();
2  ActiveMap localAM = ...;
3  RemoteActiveMap remoteAM = ...;
4  Object key = ...:
5  // Get the reflection of the "foo()" method
6  Method fooMethod =
7        MyClass.class.getMethod("foo",
8                                new Class[]{StringBuffer.class});
9  // Get the reflection of the "call" method
10 Class activeMapClass = ActiveMap.class,
11 Method callMethod =
12    activeMapClass.getMethod("call",                              // method name
13                             new Class[]{Object.class,            // key
14                                         Object[].class,          // args
15                                         Future.class});          // Future
16 // Parameter of the the "foo" method
17 Object[] fooParam = new Object[] {new StringBuffer("Dummy")};
18 // Parameter of the remote call
19 Object[] remoteCallParam = new Object[] {fooMethod,
20                                          fooParam,
21                                          new Future(...)};
22 // Parameter of the local call
23 Object[] localCallParam = new Object[] {callMethod,
24                                         remoteCallParam,
25                                         new Future(...)};
26 activeMap.call(key, callMethod, localCallParam);
```

L'utilisation de la méthode `call()` est déjà fastidieuse en raison de l'utilisation du mécanisme réflexif pour spécifier la méthode, l'encapsulation alourdit le code considérablement. Cependant, si ce problème peut sembler important de prime abord, il sera réglé avec les mécanismes de transparence que nous présenterons au chapitre §16 p179. En outre, la gestion du résultat est délicate puisqu'il faut jongler avec deux objets *futurs*.

Mais le principal inconvénient de cette solution est sa performance. Chaque appel de méthode complètement asynchrone requiert une recherche dans la table de hachage associée à chaque conteneur. Si la recherche à distance peut être tolérée (elle est de toute façon moins coûteuse que la communication), il n'en est pas de même pour la version locale.

Enfin, l'autre problème provient de l'utilisation des *stored object*s. Nous avons vu la classe `StoredObjectReference` qui représente un objet stocké. Dans le mécanisme à base de conteneurs actifs locaux, pour communiquer avec un *stored object* distant de manière complètement asynchrone, il faut passer par le *stored object* local qui représente le *proxy* du conteneur actif distant. Cela ne facilite pas la manipulation

des *stored objects* distants.

13.2.3 Conclusion

Si l'utilisation des conteneurs actifs locaux offre une solution élégante à notre problème d'asynchronisme partiel, il est néanmoins déraisonnable pour des raisons conceptuelles et d'efficacité. Le problème conceptuel, lié à la manipulation des *stored objects* à l'aide d'une instance de classe représentant une référence distante nous a amené à la solution adoptée aujourd'hui : la notion de *référence asynchrone*

Chapitre 14

Proposition : Les références asynchrones

En recherchant des solutions à l'expression de la concurrence dans un programme utilisant le concept de conteneur actif, nous avons pris conscience du rôle particulier que jouent les instances de la classe `StoredObjectReference` vue en §10.4 p111 : elle est – comme son nom l'indique – une référence sur un objet contenu dans un conteneur actif éventuellement distant. Quelle est donc la différence conceptuelle entre une instance de cette classe et l'objet réel qu'elle référence ? Dans le cas local, les communications avec l'objet stocké peuvent être réalisées par deux moyens distincts :

- en utilisant des appels de méthodes via une référence standard Java sur l'objet stocké par l'intermédiaire d'une variable ;
- en utilisant la méthode `call()` d'une instance de la classe `StoredObjectReference` par l'intermédiaire d'une variable.

Outre l'indirection introduite par l'appel `call()`, une différence fondamentale existe entre ces deux moyens : le premier est synchrone, le deuxième est asynchrone. Par conséquent, nous allons différencier deux types de références :

- **les références synchrones :** ce sont par exemple les références standards Java ou les références RMI ;
- **les références asynchrones :** les instances de la classe `StoredObjectReference` en sont des exemples ;

Nous nous sommes alors intéressés à une généralisation du concept de référence asynchrone :

Définition 14.0.1 (Référence asynchrone)
Une référence asynchrone sur un objet permet l'invocation asynchrone de toutes les méthodes de l'objet référencé.

Nous allons voir dans la suite les caractéristiques des références asynchrones. Mais avant, quelques notations nous serons utiles.

14.1 Notations

Notation 14.1.1 (Références)
Si `i` *est une instance de classe* `C`, `R(i : C)` *est une référence synchrone ou asynchrone sur* `i`, `SR(i : C)` *est une référence synchrone sur* `i` *et* `AR(i : C)` *est une référence asynchrone sur* `i`.

On note `v : T = R(i : C)` *pour définir la variable* `v` *déclarée de type* `T` *qui référence l'instance* `i` *de la classe* `C`.

Considérons le code ci-dessous :

```
1 MyClass myObject1 = new MyClass();
2 MyClass myObject2 = myObject1;
```

La variable `myObject1` est de type `MyClass` et référence l'instance (qui n'a pas de nom) de la classe `MyClass`. Si cette instance est appelée `i`, on notera

$$\texttt{myObject1} : \texttt{MyClass} = \texttt{R}(\texttt{i} : \texttt{MyClass})$$

Naturellement, on a aussi

$$\texttt{myObject2} : \texttt{MyClass} = \texttt{R}(\texttt{i} : \texttt{MyClass})$$

Nous utiliserons aussi la notation allégée $\texttt{x} = \texttt{R}(\texttt{i})$ lorsque le type déclaré de la variable `x` et la classe de l'instance `i` n'ont pas d'importance.

14.2 Égalité

Les références, qu'elles soient synchrones ou non doivent garantir la propriété suivante :

Définition 14.2.1 (Egalité)
Soit $\texttt{r1} = \texttt{R}(\texttt{i1})$ *et* $\texttt{r2} = \texttt{R}(\texttt{i2})$, *alors* $(\texttt{r1} == \texttt{r2}) \Leftrightarrow (\texttt{i1} == \texttt{i2})$

Considérons l'exemple ci-dessous :

```
1 Integer a = new Integer(5); // (1) création de l'instance nommée i
2 Integer b = new Integer(5); // (2) création de l'instance nommée j
3 Object o = a;
```

Si l'on note `i` l'instance de `java.lang.Integer` créée par l'instruction (1) et `j` celle créé par l'instruction (2), on a :

$$\begin{aligned} \texttt{a} : \texttt{Integer} &= \texttt{R}(\texttt{i} : \texttt{Integer}), \\ \texttt{b} : \texttt{Integer} &= \texttt{R}(\texttt{j} : \texttt{Integer}), \\ \texttt{o} : \texttt{Object} &= \texttt{R}(\texttt{i} : \texttt{Integer}) \end{aligned}$$

Évidemment, on a ($\texttt{a} != \texttt{b}$) même si `a.equals(b)` et `b.equals(a)` retourne `true`. Par contre, ($\texttt{a} == \texttt{o}$) alors que leur type déclaré sont différents.

14.3 Sémantique des appels de méthodes

Les appels de méthodes s'expriment toujours par l'intermédiaire d'une référence. Certaines entorses à cette règle sont autorisées dans la syntaxe du langage Java mais ce sont des simplifications syntaxiques implicites. Par exemple au sein d'une classe, l'expression `m()` est la simplification de `this.m()`.

14.3.1 Passage des paramètres

Dans le cas de références synchrones, la sémantique d'un appel est bien connue : appel par copie pour les types primitifs et appel par référence pour le reste[1].

Une référence asynchrone permet l'invocation asynchrone des méthodes de l'objet qu'elle référence. Quelle que soit la syntaxe de l'appel utilisée, il faut préciser la sémantique de cet appel. Pour le passage des paramètres, nous allons reprendre la sémantique de l'appel standard.

Notons ici une différence fondamentale avec la sémantique définie dans ProActive (*cf.* §9.3 p82) qui est celle de RMI (en réalité, ProActive s'appuie sur RMI), dans laquelle on distingue les types suivants :

- **Les types primitifs :** sont passés par copie ;
- **les types références :** sont passés par sérialisation.

On trouve parfois dans la documentation (RMI en particulier) trois mécanismes et non deux :

- **Les types primitifs :** sont passés par copies ;
- **les sous-types de `java.rmi.Remote` :** sont passés par référence ;
- **les autres types :** sont passés par sérialisation.

[1] On trouve parfois la sémantique par copie pour tous les types. La justification considère que la copie d'une référence est une référence qui – au même titre que la copie de pointeurs en langage C – référence le même objet.

Mais il n'en est rien. En fait, les sous-types de `java.rmi.Remote` sont sérialisés, mais le mécanisme de la sérialisation est redéfini pour éviter la copie en profondeur de l'objet distant et la remplacer par une référence distante. Pire, si une classe implémente l'interface `java.rmi.Remote` sans utiliser la sérialisation des références distantes RMI, la sémantique spécifiée n'est pas assurée.

14.3.2 Asynchronisme

Lors d'un appel, l'aspect asynchrone a une influence pour les deux parties impliquées dans l'appel :

- pour le client, l'appel n'est pas bloquant ;
- pour le serveur, l'appel doit être pris en compte pour son exécution

Définition 14.3.1 (Sémantique d'un appel asynchrone)
Lorsqu'un appel de méthode asynchrone est réalisé par l'intermédiaire d'une référence asynchrone, la thread appelante est bloquée le temps nécessaire à la prise en compte de l'appel par l'implémentation sous-jacente, à la construction d'un objet future *et à son retour. Elle n'attend* ***ni la fin*** *de l'exécution de la méthode,* ***ni même son début*** *pour continuer l'exécution à sa prochaine instruction.*

Cette sémantique a une implication importante :

> **L'exécution effective de la méthode spécifiée dans un appel asynchrone n'est pas nécessairement *concurrente* avec l'exécution de l'appelant. Cette exécution peut en outre ne *jamais* avoir lieu.**

De prime abord, cette sémantique peut sembler problématique. Si elle autorise la non-exécution d'une requête quelles sont ses garanties ? La réponse est claire : aucune ! Au même titre que l'ordonnanceur de threads Java ne garantit pas l'exécution d'une thread Java : les threads de basses priorités peuvent se trouver en situation de famine en raison de l'existence de threads de plus hautes priorités prêtes à être exécutées[2]. De même, une thread peut, selon la spécification, ne jamais obtenir un moniteur en Java (bloc ou méthode `synchronized`) si elle est en compétition avec plusieurs autre threads à chaque tentative.

Notons qu'il est possible de définir une autre sémantique :

Définition 14.3.2 (Sémantique d'un appel asynchrone non retenue)
Lorsqu'un appel de méthode asynchrone est réalisé par l'intermédiaire d'une référence asynchrone, la thread est bloquée le temps nécessaire pour démarrer[3] *la méthode,*

[2]Depuis l'abandon des *green-threads* depuis le JDK v1.3, aucune implémentation de JVM n'est conforme à la spécification des threads, excepté peut-être celle de Sun sous Solaris.

[3]Première instruction bytecode.

construire un objet future *et le retourner. Ensuite, elle peut continuer son exécution à sa prochaine instruction.*

Toutefois, cette définition ne sera pas retenue. En effet, elle rend particulièrement inefficace des utilisations d'implémentations à base de files, de piles ou de files de priorités comme expliqué dans l'exemple suivant.

Exemple 14.3.1 (Implémentation à base de file)
Considérons une implémentation de références asynchrones qui utilise une file pour enregistrer les appels de méthodes et une seule thread pour leur exécution. Supposons que n threads appelantes distinctes aient déjà effectué n appels de méthodes asynchrones sur la même référence[4]. La file contient donc $p \leq n$ entrées[5] pendant que la thread associée à la référence asynchrone exécute la requête la plus vieille.

Dans cette configuration, une $n + 1^{\text{ième}}$ thread qui effectue un appel asynchrone ne pourra exécuter sa prochaine instruction qu'après l'exécution des p méthodes.

Nous verrons toutefois que notre sémantique autorise des implémentations qui permettent à l'appelant de vérifier et même d'attendre le démarrage de sa méthode.

14.3.3 Asynchronisme et concurrence

Malgré notre spécification, il est tentant de penser qu'après un appel asynchrone, la méthode est exécutée en concurrence de celle qui est à l'origine de l'appel. Or, cela n'est pas nécessairement le cas. En reprenant l'exemple §14.3.1 p151, il est clair que l'appelant peut terminer son exécution alors que la méthode qu'il a appelé de manière asynchrone n'est même pas commencée[6]. En choisissant un autre type d'implémentation à base de threads dans laquelle un appel de méthode asynchrone est géré par une thread créée pour l'occasion, le même phénomène peut se produire malgré tout. Par exemple, si de nombreuse invocations sont effectuées, le système peut mettre un certain délai avant de créer la thread chargée de l'exécution de la méthode, délai suffisant pour que l'appelant termine son exécution.

Aussi, l'asynchronisme ne mène pas nécessairement à l'exécution concurrente de la méthode invoquée, mais il créé de la concurrence dans l'application globale.

[4]Et donc le même objet cible en vertu de la propriété d'égalité mentionnée en §14.2 p148.

[5]On suppose que seules nos n threads ont accès à notre référence asynchrone et que la file était vide au départ.

[6]En terme d'utilité d'une telle configuration, on peut imaginer que l'appelant partage l'objet *future* renvoyé avec une autre thread.

14.4 Récupération du résultat d'un appel de méthode asynchrone

Nous avons vu dans la section précédente la sémantique qui est attachée à une référence asynchrone. Puisque la thread appelante n'est pas bloquée durant l'exécution de la méthode invoquée, il faut fournir un moyen de récupérer le résultat de l'appel[7].

On peut distinguer deux grands types de mécanismes :

- **mécanisme actif :** c'est le client qui prend la décision de récupérer le résultat par scrutation (*polling*) ou attente ;
- **mécanisme passif :** un mécanisme événementiel (*callback*) prévient le client que l'appel de méthode asynchrone est terminé.

Ces deux mécanismes ayant leurs avantages et leurs inconvénients, nous offriront les deux.

14.4.1 Mécanisme actif

L'utilisation d'une *réponse anticipée* – objet *futur* – permet au client de garder une *référence historique* sur un appel de méthode passé et de l'utiliser pour scruter ou attendre la fin de son appel. Cet objet *futur* doit aussi permettre au client de vérifier que l'appel s'est bien déroulé et notamment qu'une exception n'a pas été levée.

14.4.2 Mécanisme passif

L'utilisation d'un *callback* permet au client d'enregistrer un gestionnaire événementiel qui sera averti par la plate-forme lorsque l'appel sera terminé. Le *callback* doit avoir la possibilité de récupérer le résultat, de récupérer l'exception qui a éventuellement été levée, de connaître la méthode qui a été invoquée, les arguments de cette méthode, la référence asynchrone utilisée pour cet appel et la thread qui a demandé l'appel.

Le mode d'invocation du *callback* a son importance. Si c'est la thread chargée de l'invocation qui notifie le *callback*, alors cette dernière n'est pas en mesure de répondre à une autre requête tant que le *callback* n'a pas terminé son exécution. Si

[7]Nous aurions pu contraindre l'utilisation des appels asynchrones à des méthodes de type procédurales, c'est à dire dont le type de retour est `void` comme dans le modèle à objets séparés (*cf.* §12.4.1.3 p139). Toutefois, cela ne résout pas les problèmes de gestion des exceptions (comment l'appelant est au courant), et cela limite fortement le potentiel de parallélisation des applications.

l'on perçoit immédiatement les problèmes de performance, cette sémantique possède un autre inconvénient majeur : elle est sujette aux étreintes fatales.

Pour illustrer ce problème, supposons que notre implémentation de références asynchrones utilise une politique de type FIFO – selon le modèle des objets actifs vu en §12.4.1.1 p134 par exemple. La thread responsable de l'invocation est donc la seule à prendre en charge les requêtes. Si lors de l'invocation d'un *callback*, ce dernier effectue un appel asynchrone sur la même référence, et attend le résultat, nous obtenons une étreinte fatale. La thread en charge de l'appel ne peut prendre en compte la requête du *callback* tant que ce dernier n'a pas terminé. Ce dernier n'aura terminé que lorsque cette dernière thread aura pris en compte l'invocation faite par le *callback*.

La thread chargée de l'appel ne doit donc pas réaliser elle même les notifications. La notification au *callback* se fera donc de manière asynchrone.

14.4.3 Synchronisation entre les deux mécanismes

Nous avons vu les deux mécanismes pour la récupération du résultat. L'utilisateur pouvant utiliser les deux en même temps, il faut spécifier si les effets de bords du *callback* doivent être visibles par les threads qui utilisent le mécanisme actif.

Par conséquent, lorsque l'objet *futur* indique que l'appel est terminé, les effets de bords du *callback* doivent être visibles.

14.5 Annulation des appels asynchrones

La gestion de la concurrence dans une application mène naturellement à des problèmes d'annulation[8]. Dans l'API des threads POSIX, nous avons déjà vu les deux *types d'annulation*[9] (*cf.* §5.2.3.1 p58) :

- annulation *collaborative* ;
- annulation *autoritaire* ;

Nous ne proposerons pas de mécanismes d'annulation de type *autoritaire* en raison des difficultés qu'elles posent au programmeur ; seule l'annulation de type *collaborative* sera considérée.

Selon la sémantique des appels asynchrones que nous avons définie en §14.3.2 p150, nous devons examiner trois cas pour fournir un mécanisme d'annulation :

- l'appel n'est pas commencé ;

[8]Malgré tout, peu de travaux apportent une solution.

[9]Que nous avons aussi appelé *types d'interruptions.*. Toutefois, cette partie faisant régulièrement référence au langage Java, nous préférerons utiliser le terme d'annulation, pour éviter la confusion avec la méthode `java.lang.Thread.interrupt()` qui est une annulation de type *collaborative*.

- l'appel est en cours d'exécution ;
- l'appel est terminé.

Nous allons définir une terminologie adaptée à ces trois cas.

Définition 14.5.1 (Etat d'un appel asynchrone)
Un appel est ***commencé*** *lorsqu'une thread a exécuté la première instruction de la méthode spécifiée. Cette thread sera nommée* thread d'appel *par opposition à* thread appelante*, celle qui a demandé l'appel.*

Un appel est ***annulable*** *s'il n'est pas commencé. L'annulation d'un appel asynchrone revient à empêcher l'exécution d'une méthode c'est à dire à interdire le commencement de l'appel.*

Un appel est considéré comme ***terminé*** *lorsqu'il est commencé, non interrompu et que la thread d'appel a atteint l'instruction suivant la dernière instruction*[10] *de la méthode spécifiée.*

Un appel est ***interruptible*** *s'il est commencé et non terminé. L'interruption d'un appel asynchrone revient à effectuer une opération d'annulation coopérative*[11] *sur la thread d'appel. Après une opération d'interruption, l'appel est dans l'état interrompu, mais non terminé.*

Notons que la nature de la terminaison (`return` ou `throw`) de la méthode n'a pas d'importance ici. Une méthode qui a levé une exception est considérée terminée au même titre qu'une méthode qui a retourné un résultat.

De par la nature dynamique et non déterministe de la programmation concurrente, il n'est pas possible de garantir le succès des opérations d'annulation et d'interruption des appels asynchrones. En effet, entre la prise en compte de la requête d'annulation et son exécution, il y a nécessairement un délai, suffisant pour que l'appel passe dans l'état commencé interdisant de fait son annulation. Il en est de même de l'opération d'interruption, l'appel peut passer à l'état terminé avant que la demande d'interruption n'ait abouti. La thread peut de toute façon refuser la requête d'interruption (nous ne fournissons pas d'annulation autoritaire).

Par conséquent, l'API devra permettre de vérifier le succès d'une opération d'annulation et d'interruption. Par contre, après un tel succès, aucun événement n'est envoyé à un callback ; l'appel n'étant pas considéré comme terminé (dans le cas d'une annulation, il n'a peut-être même pas commencé !).

[10] Qui n'est pas nécessairement `return` en Java, puisqu'une clause `finally` peut être exécutée ensuite.

[11] C'est à dire en Java, à invoquer la méthode `java.lang.Thread.interrupt()`.

14.6 Gestion des exceptions

Gérer les exceptions dans un paradigme d'appels de méthodes asynchrones est un problème délicat dont la résolution nécessite des considérations syntaxiques.

Dans un cadre totalement transparent comme celui des objets actifs ou celui des objets séparés (*cf.* §12.4.1 p134), il n'y a pas de solution simple : l'appel étant syntaxiquement identique à un appel synchrone, la thread appelante peut se trouver en dehors du bloc `catch{}` lorsqu'une exception est levée par la thread d'appel.

Par conséquent, il semble nécessaire de rendre un appel asynchrone syntaxiquement différent d'un appel synchrone. Nous verrons que la transparence des appels asynchrones est un problème majeur en §16 p179. Les deux mécanismes utilisables pour la récupération du résultat vus en §14.4 p152 permettent à la thread appelante de connaître l'exception qui a éventuellement été levée. Cependant, en ce qui concerne la sémantique d'un appel asynchrone lors de la levée d'une exception dans la thread d'appel on peut distinguer, entre-autres, les mécanismes suivants :

- interdire l'exécution de tout autre méthode sur l'objet cible tant qu'un accusé, émis par une entité capable de gérer les exceptions, n'a pas été reçu ;
- interrompre la thread appelante de manière à ce qu'elle soit informée au plus vite du mauvais déroulement de son appel.

Nous laisserons les implémentations libres de fournir les mécanismes les plus appropriés.

14.7 Opérations groupées

Il est tentant de fournir des opérations globales sur les références asynchrones du type *attendre la terminaison de tous les appels* ou *annulation de tous les appels* par exemple. Cependant, si la sémantique de ces appels est claire, quel est leur intérêt pour une thread appelante ? Une thread appelante peut vouloir interrompre tous *ses* appels, en aucun cas tous *les* appels qui ont été effectués sur un objet. La différence est de taille surtout dans le cas de références asynchrones distantes pour lesquelles plusieurs clients potentiellement distribués effectuent des appels de méthodes asynchrones. Dans ce cadre, une méthode d'annulation globale permettrait d'annuler tous les appels y compris ceux réalisés par des clients distants dont la thread appelante ne soupçonne pas l'existence.

Nous ne fournirons donc pas cette fonctionnalité. Par contre, un *callback* particulier permettant des opérations sur un ensemble d'appels est fourni par la classe `ResultsGrouper` que nous ne présenterons pas ici[12]. La différence réside dans le

[12]Le lecteur peut se référer au fichier `src/mandala/rami.impl/ResultsGrouper.java` pour étudier le code source qui est très simple.

fait que la thread appelante pourra définir précisément les appels qu'elle souhaite annuler par exemple.

14.8 Bilan

Nous avons défini notre concept de références asynchrones. Nous avons vu les propriétés de ce concept : l'égalité, la sémantique des appels de méthodes, les mécanismes de récupération du résultat, l'annulation et l'interruption d'appels, et la gestion des exceptions. Nous allons présenter notre implémentation de ce concept dans le chapitre suivant.

Chapitre 15

Implémentation en Java : RAMI

Dans la mesure où Java ne fournit pas de références asynchrones et qu'il n'est pas raisonnable de modifier la machine virtuelle et le langage Java[1], nous avons décidé de réaliser une implémentation 100% Java, donc sans modification ni du langage, ni de la machine virtuelle.

15.1 Interfaces

Nous avons défini les deux notions de référence et de référence asynchrone par deux interfaces[2] données en §15.1 p157 et §15.2 p158.

```
1 public interface Reference{
2     Object getObject();
3 }
```

Prog. 15.1 – L'interface Reference

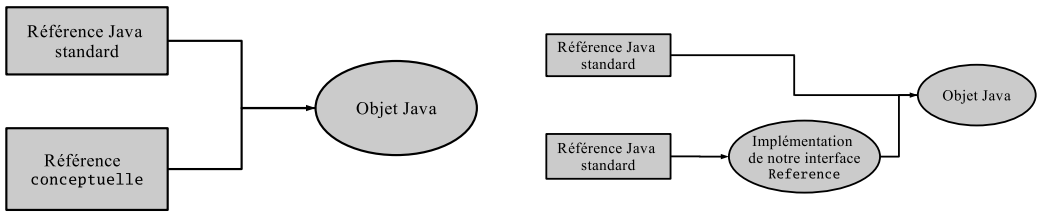
L'interface `Reference` présentée en Prog. 15.1 p157 offre simplement la possibilité de récupérer une référence standard sur l'objet référencé. L'intérêt de cette interface s'il n'est pas immédiat, est d'assurer la compatibilité entre types de références différents. L'idée est de permettre l'utilisation dans le même programme de plusieurs sortes de références (les références du paquetage `java.lang.ref` et nos références asynchrones par exemple). Par ailleurs, dans un souci d'extensibilité et d'intégration

[1] Notre équipe est trop petite pour se lancer dans une telle aventure.

[2] Comme mentionné en §10.1 p95, nous avons décidé d'utiliser la méthode de conception par interfaces au maximum.

éventuelle dans un JDK future, nous reprenons en partie l'API de la classe abstraite `java.lang.ref.Reference`.

Si conceptuellement nos références sont similaires aux références Java standard, du point de vue de l'organisation mémoire, les choses sont différentes comme l'illustre le schéma FIG. 15.1 p158.



(a) Le concept de référence

(b) L'implémentation de ce concept en Java

FIG. 15.1 – Implémentation du concept de référence.

L'interface `AsynchronousReference` est la pierre angulaire de notre concept. Elle définit la méthode `call()` qui permet l'invocation asynchrone de n'importe quelle méthode publique[3] de l'objet référencé.

```
1 public interface AsynchronousReference extends Reference{
2     FutureClient call(Method method,
3                       Object[] args,
4                       Callback callback);
5 }
```

PROG. 15.2 – L'interface AsynchronousReference

15.1.1 Futures

Dans un appel asynchrone, deux entités sont à considérer : la thread appelante et la thread d'appel. De ce point de vue et par analogie au modèle à base de passage de messages dans un système distribué, nous appellerons la thread appelante le *client* et la thread d'appel le *serveur*. Nous allons définir les besoins – en terme de méthodes – qu'ont un client pour utiliser le résultat d'un appel asynchrone, et un serveur pour l'exécution de la méthode.

[3]Les classes qui implémenteront l'interface `AsynchronousReference` n'auront pas nécessairement le droit d'invoquer les méthodes non publiques des objets qu'elles référenceront.

15.1.1.1 Besoins du client

Nous allons définir une interface qui répondra à chacun des besoins spécifiques :

`InvocationInfo` : contient les méthodes `getAsynchronousReference()`, `getMethod()`, `getArgs()` et `getCallerThread()` qui permettent respectivement de retrouver la référence cible de l'appel, la méthode invoquée, les arguments spécifiés et la thread appelante ;

`InvocationObserver` : contient les méthodes `isStarted()`, `isCancelled()`, `isInterrupted()`, `isResultAvailable()`, qui permettent de déterminer l'état de l'appel ;

`InvocationEventsWaiter` : contient les méthodes `waitUntilStarted()`, `waitForResult()`, `waitUntilResultAvailable()` qui permettent l'attente (éventuellement avec un délai) d'un événement particulier ;

`MethodResult` : contient les méthodes `getReturnedResult()` et `getException()` pour récupérer le résultat qu'il soit un retour de méthode ou une exception.

Ces interfaces sont directement héritées par l'interface `FutureClient` qui est le type de retour de la méthode `call()` de l'interface `AsynchronousReference`.

Ce découpage fin permet une identification claire des fonctions de l'interface et une documentation simplifiée. Le résultat de certaines méthodes n'est pas valide – et on dira que ces méthodes sont *non-sûres* – tant qu'une condition n'est pas vraie. C'est le cas en particulier de l'ensemble des méthodes de l'interface `MethodResult` dont les valeurs de retour ne sont pas significatives tant que la méthode `InvocationObserver.isResultAvailable()` retourne la valeur `false`. Au contraire, l'ensemble des méthodes des interfaces `InvocationInfo`, `InvocationObserver` et `InvocationEventsWaiter` sont *sûres*.

15.1.1.2 Besoins du serveur

Nous allons procéder de la même manière que précédemment, en spécifiant les interfaces requises par le serveur :

`InvocationInfo` : nous retrouvons cette interface qui permet au serveur de récupérer les informations nécessaires à l'appel (méthode et arguments) ;

`ResultUpdater` : contient les méthodes `setStarted()` et `setResult()` qui permettent de mettre à jour l'état de l'appel (commencé, terminé) et son résultat ;

On retrouve donc des informations nécessaires aux deux entités (méthode et arguments), et des besoins propres pour mettre à jour le résultat. L'interface `FutureServer` est donc définie par symétrie avec `FutureClient`, elle hérite des deux interfaces `InvocationInfo` et `ResultUpdater`.

15.1.2 Callbacks

Nous avons vu en §14.4 p152 qu'un mécanisme passif pour la récupération du résultat devait être proposé. Ce mécanisme événementiel utilise un objet *callback* dont l'unique méthode `done()` sera invoquée lorsque le résultat de l'appel est disponible. Nous proposons donc l'interface `Callback` présentée en PROG. 15.3 p160.

```
1 public interface Callback {
2     void done(InvocationInfo invocationInfo,
3               MethodResult methodResult);
4 }
```

PROG. 15.3 – L'interface `Callback`

15.1.3 Annulation

Nous devons fournir les moyens d'annuler ou d'interrompre un appel de méthode asynchrone (*cf.* §14.5 p153). A cet effet, nous proposons l'interface `Cancelable` définie par le code PROG. 15.4 p160.

```
1 public interface Cancelable {
2     boolean cancel();
3     boolean interrupt();
4 }
```

PROG. 15.4 – L'interface `Cancelable`

Cette interface est aussi héritée par `FutureClient`.

Ainsi, pour demander l'annulation ou l'interruption d'un appel asynchrone, l'utilisateur, après avoir récupéré son objet `future` de type `FutureClient`, utilise l'instruction :

```
1 boolean ok = future.cancel() || future.interrupt();
```

15.1.4 Implication sur les interfaces de JACOb

Nous avons vu que les instances de la classe `StoredObjectReference` présentée en §10.4 p111 sont des références asynchrones sur des *stored objects*. Cette classe doit donc implémenter l'interface `AsynchronousReference`. Sa méthode `call()` possède

donc la même signature que celle de l'interface `AsynchronousReference`. En outre, nous adopterons la notation suivante :

Notation 15.1.1 (Références sur les *stored objects*)
Si `i` *est un* stored object, *alors* `SOR(i)` *est une référence sur* `i` *et* `SOR(i)`≡`R(i)`. *Si le conteneur actif associé à un* stored object *est connu pour être* distant *on parlera de* stored object distant *et on notera* `RSOR(i)` *pour désigner sa référence.*

L'entité qui réalise un appel asynchrone sur un *stored object* n'est pas l'instance de la classe `StoredObjectReference`. En réalité, c'est le conteneur actif représenté par l'interface `ActiveMap` qui effectue l'appel. C'est donc lui qui joue le rôle du serveur. Nous avons présenté cette interface en Prog. 10.2 p97 ; elle va être modifiée pour que la méthode `call()` prenne en argument la clé et un objet `FutureServer` qui permet à la fois de connaître la méthode à invoquer (avec ses arguments) et de mettre à jour l'état de l'appel. En outre, la méthode `call()` retourne désormais un objet de type `Cancelable` permettant d'annuler un appel. La nouvelle interface `ActiveMap` est présentée en Prog. 15.5 p161.

```
1 public interface ActiveMap extends Map {
2    Cancelable call(Object key, FutureServer futureServer);
3 }
```

Prog. 15.5 – L'interface définitive `ActiveMap`.

15.2 Implémentations

Dans cette section, nous allons présenter succinctement les deux implémentations de notre interface `AsynchronousReference` : une version locale, et une version basée sur le concept de conteneur actif.

15.2.1 Singleton

L'interface `AsynchronousReference` ne suffit pas en elle-même à simuler la notion de référence Java. Il faut en effet que nos références possèdent la caractéristique mentionnée en §14.2 p148 : l'égalité. Cette caractéristique ne peut être imposée par l'interface qui ne contient pas de constructeur. Une classe abstraite n'est pas non plus d'une grande utilité puisque les constructeurs ne sont pas hérités. Aussi, est-il nécessaire de faire confiance à l'implémentation pour qu'elle implémente de manière correcte cette caractéristique.

Pour faciliter son développement nous proposons un squelette de code qui fait appel à une classe particulière appelée `SingletonGiver`. Cette classe permet de récupérer un objet appelé *singleton* – en référence au *design pattern* du même nom de la bande des quatre [80] – associé à un autre objet ou de le créer si une telle association n'existe pas. Elle utilise une *table canonique* du même type que `java.util.WeakHashTable` mais basée sur les valeurs et non sur les clés. Ainsi, pour une implémentation de l'interface `AsynchronousReference`, le code à écrire est donné en PROG. 15.6 p163.

Dans ce squelette de code, la méthode `readResolve()` est implémentée de manière à remplacer l'instance déserialisée par celle qui est éventuellement déjà associée à l'objet. Si ce n'est pas le cas, l'instance déserialisée devient le singleton, sinon, il devient candidat au ramasse-miettes. La classe `SingletonGiver` utilise (en interne par le biais de la classe `WeakValuesMap`) des associations à des références faibles afin d'assurer l'élimination des singletons non référencés et ainsi éviter des problèmes de fuite mémoire.

Le singleton est indispensable pour obtenir une simulation correcte de la notion de référence. Toutefois, les effets de bords de cette caractéristique sont nombreux.

15.2.1.1 Impact sur les performances

La classe `SingletonGiver` n'est pas seulement utilisée pour les références asynchrones. Elle est en fait utilisée pour éviter la création de multiples instances identiques sur le plan sémantique. Par exemple, dans la méthode `call()` de l'interface `AsynchronousReference`, le premier argument est de type `Method`. Généralement, les mêmes méthodes sont invoquées plusieurs fois. Aussi, l'utilisation d'une table canonique évite de multiples instanciations de la classe `Method` qui désigne la même méthode. Notons que nous n'utilisons pas la classe `java.lang.reflect.Method` qui n'est pas sérialisable, et proposons notre propre implémentation `SerializableMethod` à cet effet. Cette classe utilise le squelette de code §15.6 p163.

L'impact d'une table canonique dans le cas général sur les performances globales d'une application est difficile à mesurer. Les interactions avec le ramasse-miettes et les optimisations réalisées par le JIT dépendent d'une JVM à l'autre. Les facteurs que nous pouvons influencer et étudier sont :

- le nombre de requêtes (à `new` et à `getInstance()`) ;
- la taille des objets ;
- le pourcentage d'instances égales par rapport au nombre d'instances totale.

Pour nos mesures, nous avons considéré le programme de test suivant : on dispose d'une classe qui est censée représenter un singleton (par exemple une implémentation de notre interface `AsynchronousReference` ou notre classe `SerializableMethod`). Chaque instance de cette classe est associée à un objet particulier qu'elle sert (par

```
1  // Object is the object to get an asynchronous reference on ParamJ is a
2  // parameter my own implementation needs to  create an asynchronous reference
3  protected MyAsynchronousReference(Object object,
4                                    Param1 param1,
5                                    ...,
6                                    ParamN paramN) { ... }
7
8  private static final SingletonGiver singletonGiver =
9          new SingletonGiver(new SingletonGiver.Factory() {
10                 // Implements my singleton policy
11                 public Object newInstance(Object object, Object[] args) {
12                     // if args.length == 1 it means singleton is used in
13                     // deserialization (see readResolve() below) and that
14                     // args[0] contains the reference to return.
15                     if (args.length == 1) {
16                         return args[0];
17                     }
18
19                     if (args.length == N) {
20                         return new MyAsynchronousReference(object,
21                                                            param1,
22                                                            ...,
23                                                            paramN);
24                     }
25                     throw new Error("Wrong number of arguments!");
26                 }
27         });
28
29 public static MyAsynchronousReference getInstance(Object object,
30                                                    Param1 param1,
31                                                    ...,
32                                                    ParamN paramN) {
33     return (MyAsynchronousReference)
34             singletonGiver.getInstance(object,
35                                        new Object[]{param1,
36                                                     ...
37                                                     paramN});
38 }
39
40 // Warning : Singleton must be respected !
41 // readResolve() help us in this way !
42 private Object readResolve() throws ObjectStreamException {
43 // Do not create a MyAsynchronousReference with 'new' as in
44 // getInstance(), use 'this' instead
45         return singletonGiver.getInstance(object,
46                                           new Object[]{this});
47 }
```

Prog. 15.6 – Squelette du code pour l'implémentation de la caractéristique d'égalité des références.

exemple l'objet retourné par `AsynchronousReference.getObject()`, ou la méthode retournée par `SerializableMethod.getMethod()`). Pour les besoins de l'étude, cette classe dispose d'un constructeur déclaré `public`[4]. Le but est de déterminer les paramètres – le nombre de requêtes et la taille des objets – qui favorisent l'utilisation de la table canonique.

Une instance de notre classe de test contient donc deux champs : un tableau d'octets (`byte[]`) utilisé pour paramétrer sa taille et une instance de la classe `Integer`. Nous cherchons à mesurer le rapport entre le temps nécessaire à l'instanciation de n objets et celui nécessaire à la récupération de n références (Java standard) par le biais de notre classe `SingletonGiver`. La classe `SingletonGiver` évite la création d'instances égales entre-elles, elle fait donc en quelques sorte office de cache : la méthode `SingletonGiver.getInstance()` utilisée pour la récupération d'une référence créée éventuellement une instance s'il n'y a pas d'association trouvée avec l'entier spécifié dans la table canonique.

L'autre paramètre que nous faisons varier est la taille des objets. Il est évident que pour les petits objets le "cache" n'a pas d'intérêt. Aussi, nous avons pris comme étalon la taille moyenne d'un objet Java qui est de l'ordre de 30 octets [216].

Pour un nombre d'appels n et une taille t fixée, on note $T_i(n,t)$ le temps nécessaire à la récupération de n références sur des objets de taille t par instanciation. De même, $T_c(n,t)$ est le temps nécessaire à la récupération de n références sur des objets de taille t par utilisation de la classe `SingletonGiver` (et donc d'une table canonique). On cherche à mesurer le rapport :

$$R(n,t) = \frac{T_i(n,t)}{T_c(n,t)}$$

Si $R \ll 1$, alors l'utilisation de la classe `SingletonGiver` a un impact négatif sur les performances ($\frac{1}{R}$ fois moins efficace). Si $R \simeq 1$ alors l'utilisation de la classe `SingletonGiver` n'a pas d'impact significatif sur les performances. Si $R \gg 1$, l'utilisation du `SingletonGiver` offre un gain en performance très intéressant (R fois plus efficace).

Les graphiques de la figure FIG. 15.2 p166 montrent les résultats de cette étude dans les cas où 20 % et 80 % des instances sont égales pour des instances de tailles variables. En abscisse figure le nombre d'appels effectués (`new` ou `getInstance()`), en ordonnée, le rapport R précédemment défini.

Dans le premier cas, on remarque que pour les objets de taille inférieure à 90 octets, l'apport du cache est inexistant ou négligeable. Pour les objets plus gros, le cache est intéressant aux environs de 30 000 appels. Par contre, dans le deuxième cas,

[4]Dans les classes réelles, un constructeur public ne doit pas exister pour garantir la propriété du singleton.

le cache devient intéressant dès 9000 appels. Si ces valeurs semblent importantes, il faut noter que dans un programme Java, le nombre d'instances créées s'élève à plusieurs millions[5] !

En ce qui concerne RAMI, et son cadre d'utilisation (programmation concurrente et éventuellement distribuée comme nous le verrons bientôt), les applications sont conformes à la règle des 80-20 : 80% du temps s'exécute dans 20% du code. On peut donc considérer que relativement peu de méthodes sont très souvent invoquées. Par contre, leurs paramètres peuvent changer. Aussi, la création pour chaque appel d'une instance de la classe `Method` donnerait plus de travail au ramasse-miettes et ralentirait l'application. C'est la raison pour laquelle la méthode `call()` ne prend pas en paramètre une référence sur un objet de type `java.lang.reflect.Method` mais sur une version sérialisable nommée `SerializableMethod` qui est canonique grâce à l'utilisation de notre classe `SingletonGiver`.

Par contre, il peut sembler déraisonable d'utiliser 9000 références asynchrones dans la même application. Cependant cette remarque est à relativiser pour deux raisons :

- le singleton n'est pas un mécanisme prévu pour l'amélioration des performances au départ, mais pour l'implémentation de la propriété d'égalité spécifiée en §14.2 p148 ;
- le concept de références asynchrones est destiné à être utilisé de manière exclusive : toutes les références manipulées dans un programme devraient être des références asynchrones[6].

15.2.1.2 Aide apportée par le singleton

L'utilisation d'instances canoniques pour nos références asynchrones facilite l'implémentation de certaines fonctionnalités comme :

Moniteur : Le singleton permet d'utiliser une référence asynchrone comme un moniteur. Après avoir obtenu une référence asynchrone, on peut l'utiliser dans un bloc `synchronized` comme tout autre référence. Cela n'est pas le cas avec RMI par exemple, deux *stubs* différents pouvant en effet référencer le même objet distant.

[5] 1 493 518 lors de l'exécution du programme `CodeAnalyser` présenté en §20.2 p236 et 2 053 478 à la première exécution de DJFractal [211]. Les mesures ont été réalisées avec JProfiler v3.0 de ej-technologies `http://www.jprofiler.com`.

[6] Cela demande soit une grande rigueur de la part du développeur où l'utilisation d'un transformateur automatique de code qui transforme toutes les références standards Java en des références asynchrones. Un tel outil doit évidemment garantir que l'application transformée a le même comportement que l'application d'origine.

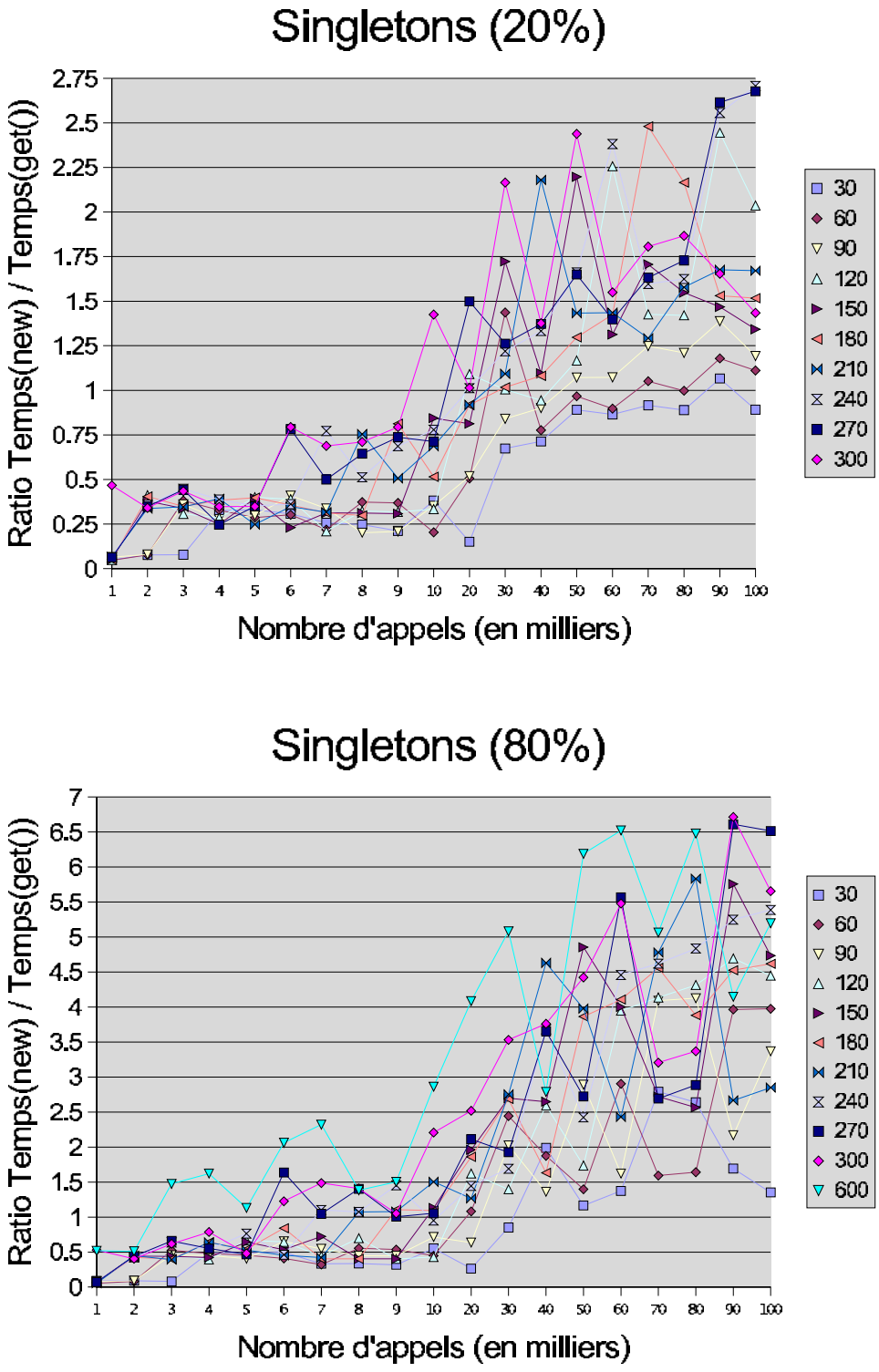


FIG. 15.2 – Utilisation de la classe `SingletonGiver` et impact sur les performances

Réservation d'objets : Le singleton facilite l'implémentation[7] d'un mécanisme de réservation qui permet à un objet d'utiliser de manière exclusive un autre objet.

Optimisation des communications : dans le cas distant, le singleton permet d'optimiser les communications en évitant une couche supplémentaire destinée à synchroniser les appels[8] et surtout en permettant de récupérer une référence locale lorsque le client est dans la même JVM que le serveur (méthode `getLocalReference()` de l'interface `mandala.jacob.remote.Remote` présentée en Prog. 10.3 p102).

15.2.2 Implémentation des *future*s

Nous pouvons proposer un code générique pour notre interface `FutureClient`. Une classe abstraite `AbstractFutureClient` est donc fournie pour factoriser un certain nombre de méthodes.

15.2.3 Implémentation locale

Nous allons présenter une implémentation locale et complètement indépendante du concept de conteneur actif de l'interface `AsynchronousReference`. Cette implémentation appelée `AsynchronousReferenceImpl` utilise le squelette de code Prog. 15.6 p163 pour assurer la propriété d'égalité vue en §14.2 p148.

Nous allons nous intéresser plus particulièrement à la méthode `call()` que nous devons implémenter. Elle doit renvoyer un objet de type `FutureClient`. Cependant, avant de retourner cet objet, l'appel doit avoir été pris en compte par une implémentation capable d'*annuler* un appel (*cf.* §14.5 p153). L'implémentation est donnée en Prog. 15.7 p168. La gestion de l'asynchronisme est ainsi déléguée à une instance particulière, référencée par le champ `policy`, qui est associée à chacune de nos références asynchrones. En paramétrant cette instance, la politique de gestion de la concurrence peut être spécialisée.

15.2.3.1 Politiques asynchrones

Cette instance, de type `AsynchronousPolicy` est une interface définie en Prog. 15.8 p168.

Mandala propose plusieurs implémentations de cette interface offrant des *sémantiques* asynchrones différentes :

[7] Locale ! Dans le cas distant, les choses sont nettement plus compliquées.

[8] En effet, si plusieurs objets représentent le même objet distant, un mécanisme doit être employé pour garantir un accés séquentiel à la couche réseaux.

```
1  public class AsynchronousReferenceImpl implements AsynchronousReference {
2
3      protected final Object object;
4      protected final AsynchronousPolicy policy;
5
6      public FutureClient call(final Method method,
7                               final Object[] args,
8                               final Callback callBack) {
9
10         final Thread caller = Thread.currentThread();
11         final FutureServer = new FutureServer(this, // target
12                                               method,
13                                               args);
14         // Delegates to policy
15         final Cancelable cancelable = policy.call(object, futureServer);
16
17         return FutureClient(this,
18                             method,
19                             args
20                             caller,
21                             callBack,
22                             cancelable);
23     }
24
25     // ***************** Singleton *****************
26     ...
27 }
```

PROG. 15.7 – Implémentation locale de la classe `AsynchronousReference`.

```
1  public interface AsynchronousPolicy{
2      Cancelable call(Object object, FutureServer futureServer);
3  }
```

PROG. 15.8 – Politiques asynchrones : l'interface `AsynchronousPolicy`.

- **sémantique concurrente :** les méthodes invoquées de manière asynchrone peuvent éventuellement s'exécuter en concurrence. Deux politiques fournissent cette sémantique :
  - **`ThreadedPolicy` :** qui implémente une sémantique concurrente de type "une thread par appel" ;
  - **`ThreadPooledPolicy` :** qui utilise un *threads pool* pour éviter le surcoût de la création des threads.
- **sémantique non-concurrente :** les méthodes invoquées de manière asynchrones ne sont jamais exécutées en concurrence. Là aussi, deux politiques sont proposées :
  - **`FifoPolicy` :** qui utilise un algorithme *first in first out* pour sélectionner la prochaine méthode à exécuter ;
  - **`RandomPolicy` :** qui sélectionne de manière aléatoire la prochaine méthode à exécuter.

Les deux types de sémantiques asynchrones sont *absolument nécessaires* pour assurer un code correct à moins de recourir à des analyses statiques de programme pour décider de la sémantique qu'il faut associer à un objet que l'on souhaite utiliser de manière asynchrone. En effet, les objets qui ne sont pas *thread-safe* ne peuvent évidemment pas être associés à une sémantique concurrente. De même, si seule une sémantique non-concurrente est disponible le développeur doit éviter de se trouver dans une situation comme celle décrite en §12.4.1.1 p135 dans laquelle des appels asynchrones récursifs indirects sont réalisés pour éviter des problèmes d'étreintes fatales. En particulier le critère d'activabilité (*cf.* §12.4.1.1 p137) permet de développer de manière sûre[9] dans le modèle à objets actifs qui utilise exclusivement des sémantiques non-concurrentes.

Remarquons toutefois que toutes les politiques mènent à des étreintes fatales. En effet, le nombre de threads maximum allouées p, est nécessairement borné :

- par le système dans le cas de `ThreadedPolicy`,
- par l'utilisateur dans le cas de `ThreadPooledPolicy`,
- ou par la politique elle-même dans le cas des sémantiques non-concurrentes dans lesquelles $p = 1$.

Une étreinte fatale survient lorsque p méthodes bloquantes ont été invoquées et prises en charge par l'implémentation de la politique. Le $p + 1$ ième appel chargé de débloquer (`notify()`) les p autres threads ne peut aboutir puisque le nombre maximal de threads autorisé est atteint et que chaque thread est en attente.

Les politiques asynchrones peuvent être ordonnées en fonction de leur sensibilité à ce type d'étreinte fatale qui est du à la borne p définie plus haut :

[9] Mais sous de fortes contraintes.

```
FIFO == Random <= ThreadPooled <= Threaded
```

Au niveau des performances, les sémantiques concurrentes apportent un degré de concurrence dans les applications qui est supérieur à celui des sémantiques non-concurrentes. En particulier, sur des machines multiprocesseurs de type SMP, les sémantiques concurrentes peuvent engendrer du parallélisme au sein du même objet.

Notons enfin qu'une politique peut être *partagée* par plusieurs références asynchrones : il suffit que la politique représentée par le champ `policy` de deux instances distinctes[10] de le classe `AsynchronousReferenceImpl` référence le même objet. Contrairement à une sémantique concurrente, une sémantique non-concurrente ne peut généralement pas être partagée : la probabilité d'aboutir à une situation d'étreinte fatale comme celle décrite précédemment est d'autant plus grande qu'elle est partagée par de nombreuses références asynchrones et que des appels bloquants sont réalisés.

15.2.3.2 Implication pour l'implémentation de l'interface `ActiveMap`

Nous avons vu en §10.1.1 p97 que la méthode asynchrone `call()` de notre conteneur actif définie par l'interface `ActiveMap` (*cf.* PROG. 15.5 p161) et implémentée dans la classe `ActiveMapImpl` (*cf.* §10.1.1 p97) créait une thread par appel. En fait, cette politique est elle aussi paramétrable en utilisant exactement les mêmes classes (`ThreadedPolicy`, `FifoPolicy`, ...). A l'instanciation de la classe `ActiveMapImpl` c'est naturellement une politique asynchrone qui doit être spécifiée.

15.2.3.3 Implémentation distante

Nous avons déjà présenté la classe `StoredObjectReference` qui devient une implémentation particulière de l'interface `AsynchronousReference` : elle délègue à un conteneur actif l'exécution de la méthode spécifiée. Dans le paragraphe précédent, nous avons vu que la politique asynchrone utilisée est celle du conteneur actif. Par conséquent, l'implémentation de la méthode `call()` présentée en PROG. 15.9 p171 fait directement appel à la méthode `call()` du conteneur actif.

Remarquons que notre classe `StoredObjectReference` n'est pas compatible avec la sémantique des appels de méthodes telle que nous l'avons définie en §14.3 p149. En effet, lorsqu'un *stored object* est distant, les paramètres de type référence passés à la méthode `call()` sont sérialisés. Nous considérons que cette disymétrie dans le modèle n'est pas un réel problème pour deux raisons :

- les développeurs sont habitués à cette disymétrie qui existe déjà dans RMI ;

[10] Nécessairement distinctes en vertu de la propriété d'égalité décrite en §14.2 p148.

```
1  public class StoredObjectReference implements AsynchronousReference{
2      protected Object key;
3      protected ActiveMap activeMap;
4
5      // **** Implementation of AsynchronousReference ****
6      public FutureClient call(final Method method,
7                               final Object[] args,
8                               final Callback callBack) {
9
10         final Thread caller = Thread.currentThread();
11         final FutureServer = new FutureServer(this, // target
12                                               method,
13                                               args);
14         // Delegates to activeMap
15         final Cancelable cancelable = activeMap.call(key, futureServer);
16
17         return FutureClient(this,
18                             method,
19                             args
20                             caller,
21                             callBack,
22                             cancelable);
23     }
24
25     // put(), get(), remove(), as before
26     ...
27
28      // *************** Singleton ***************
29     ...
30 }
```

Prog. 15.9 – La classe `StoredObjectReference` (définitive)

- en utilisant les conteneurs actifs comme un modèle mémoire (*cf.* §9.11.1 p91) dans lequel il n'y a que des *stored objects*, ce problème n'apparaît plus.

15.3 Asynchronisme complet : chaînage de références

Nous nous sommes intéressés à l'expression de la concurrence dans les applications orientées objets en raison du problème d'asynchronisme uniquement *côté serveur* de la méthode `ActiveMap.call()` (*cf.* §12 p123). Nous avons présenté deux implémentations distinctes de notre concept de référence asynchrone. Il est temps de résoudre notre problème initial. En §13.2 p142, nous avons proposé une solution élégante en chaînant un conteneur actif local et un conteneur actif distant. Le premier fournit l'asynchronisme côté client, le second l'asynchronisme côté serveur[11]. Nous allons reprendre cette idée mais avec un chaînage de références asynchrones. Une première référence asynchrone locale est utilisée pour référencer un *stored object*. Évidemment, nous retrouvons le problème syntaxique de la proposition à base de conteneurs. Une solution va être trouvée en utilisant le concept de *paire de références* très inspiré des listes du langage LISP.

15.3.1 Paire de références

En reprenant la notation établie en §14.1 p148, on définit une paire de références de la façon suivante :

Définition 15.3.1 (Paire de références)
Soit $\mathtt{r_1} = \mathtt{R_{T_1}}(\mathtt{i})$ *une référence d'un certain type* $\mathtt{T_1}$ *(asynchrone et FIFO par exemple) et* $\mathtt{r_2} = \mathtt{R_{T_2}}(\mathtt{r_1}) = \mathtt{R_{T_2}}(\mathtt{R_{T_1}}(\mathtt{i}))$*, une référence de type* $\mathtt{T_2}$ *sur la référence* $\mathtt{r_1}$ *(réification).*

Une paire de références `p` *sur* `i` *est telle que les appels de méthodes réalisés à travers* `p` *sur* `i` *sont encapsulés pour être acheminés au travers des références* $\mathtt{r_2}$ *puis* $\mathtt{r_1}$*. On notera :*

$$\mathtt{p} =< \mathtt{r_2}, \mathtt{r_1} > (\mathtt{i}) \equiv \mathtt{R}(\mathtt{i})$$

Lorsque seul le type des références utilisé par la paire a une importance, on se contentera d'écrire :

$$\mathtt{p} =< \mathtt{R_{T_2}}, \mathtt{R_{T_1}} > (\mathtt{i})$$

[11] Rappelons que l'asynchronisme côté serveur est indispensable pour garantir la prise en compte de l'appel.

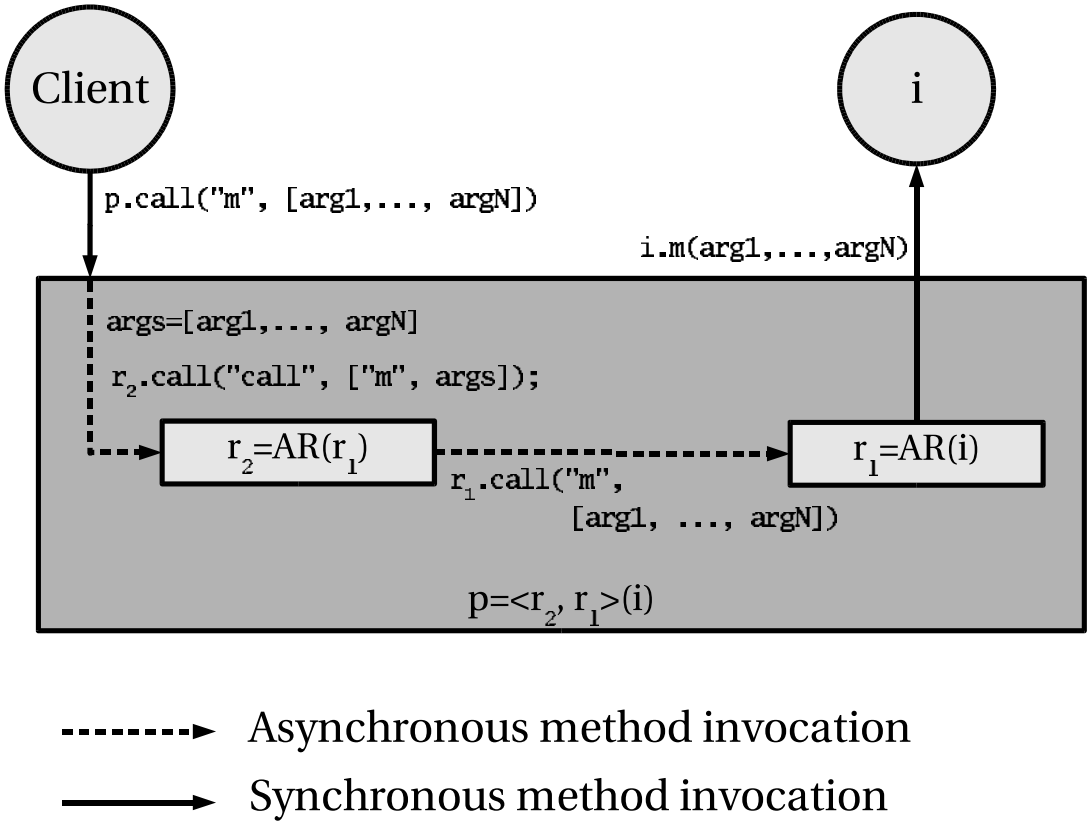


Fig. 15.3 – Appels induits par un appel à la méthode `call()` d'une paire de références.

Autrement dit, une paire de références peut être considérée comme une référence standard (au sens RAMI défini en §15.1 p157). Dans le cas des références asynchrones, l'appel à la méthode `call()` d'une paire de références induit deux appels succesifs comme l'illustre la figure Fig. 15.3 p173.

Notons que le chaînage n'est pas limité à deux références comme le montre l'exemple suivant :

Exemple 15.3.1 (Chaînage multiple)
Supposons que l'on cherche à accéder à un objet `i` distant non *thread safe*. L'objet doit donc devenir un *stored object distant*, `RSOR(i)` en étant inséré dans un conteneur actif distant. Si l'on ne connaît pas la sémantique asynchrone (*cf.* §15.2.3.1 p167) utilisée par le conteneur, le *stored object* sera en réalité une référence asynchrone locale sur `i`, associé à une sémantique non concurrente, une politique *First in First Out* par exemple et que nous noterons $\mathtt{AR}_{\mathtt{FIFO}}(\mathtt{i})$. On fabrique ensuite une paire de références : $< \mathtt{RSOR}, \mathtt{AR}_{\mathtt{FIFO}} > (\mathtt{i})$. L'utilisation d'une `RSOR` mène à un asynchronisme côté serveur. Aussi, on chaîne cette dernière référence asynchrone (qui est une paire),

avec une référence asynchrone locale pour obtenir un asynchronisme complet. Le type de sémantique n'a pas d'influence ici puisque le *stored object* est de toute façon associé à une sémantique non-concurrente. Par contre, la politique utilisée a une importance pour le client, puisqu'elle conditionne les performances de l'invocation. Une politique basée sur un *threads pool* est donc recommandée. Nous reviendrons sur ce point en §15.3.3 p177. Enfin, une dernière paire de références permet de considérer le tout comme une simple référence asynchrone. En résumé, on a donc :

$$
\begin{array}{rcll}
\texttt{r1} & = & \texttt{AR}_{\texttt{FIFO}}(\texttt{i}) & \textit{Politique FIFO} \\
\texttt{r2} & = & \texttt{RSOR}(\texttt{r1}) = \texttt{RSOR}(\texttt{AR}_{\texttt{FIFO}}(\texttt{i})) & \textit{Côté serveur} \\
\texttt{s} & = & < \texttt{RSOR}, \texttt{AR}_{\texttt{FIFO}} > (\texttt{i}) & \textit{Paire} \\
\texttt{s} & \equiv & \texttt{AR}(\texttt{i}) & \\
\texttt{r3} & = & \texttt{AR}_{\texttt{ThreadPooled}}(\texttt{s}) = \texttt{AR}_{\texttt{ThreadPooled}}(< \texttt{RSOR}, \texttt{AR}_{\texttt{FIFO}} > (\texttt{i})), & \textit{Côté client} \\
\texttt{t} & = & < \texttt{AR}_{\texttt{ThreadPooled}}, < \texttt{RSOR}, \texttt{AR}_{\texttt{FIFO}} >> (\texttt{i}), & \textit{Paire} \\
\texttt{t} & \equiv & \texttt{AR}(\texttt{i}). &
\end{array}
$$

Pour faciliter la notation, nous utiliserons la simplification syntaxique suivante :

$$< \texttt{AR}_{\texttt{T}_1}, \texttt{AR}_{\texttt{T}_2}, \ldots, \texttt{AR}_{\texttt{T}_n} > (\texttt{i}) \equiv < \texttt{AR}_{\texttt{T}_1}, < \texttt{AR}_{\texttt{T}_2}, \ldots, < \texttt{AR}_{\texttt{T}_{n-1}}, \texttt{AR}_{\texttt{T}_n} > \ldots > (\texttt{i})$$

Donc, dans l'exemple précédent, la référence `t` sera noté :

$$\texttt{t} = < \texttt{AR}_{\texttt{ThreadPooled}}, \texttt{RSOR}, \texttt{AR}_{\texttt{FIFO}} > (\texttt{i})$$

Les paires de références sont compatibles avec la solution à base de conteneurs locaux de notre problème initial. Cependant, au lieu d'utiliser le *proxy* d'un conteneur distant comme un *stored object* d'un conteneur local, on préférera stocker une référence distante, `RSOR`, sur un *stored object distant* `i` dans un conteneur actif local. On obtient donc la configuration suivante : $\texttt{SOR}(\texttt{RSOR}(\texttt{i}))$. En utilisant une paire de références, on obtient : $< \texttt{SOR}, \texttt{RSOR} > (\texttt{i}) \equiv \texttt{AR}(\texttt{i})$. Par conséquent, l'utilisation des conteneurs actifs comme modèle mémoire (*cf.* §9.11.1 p91) dans lequel seuls des *stored object* sont manipulés (via des `SOR`), reste envisageable. Le problème de performance est néanmoins non résolu par cette solution. Il vaut donc mieux dans ce cas utiliser une référence asynchrone locale comme dans l'exemple précédent :

$$< \texttt{AR}_{\texttt{ThreadPooled}}, \texttt{RSOR} > (\texttt{i})$$

Nous proposons l'interface `AsynchronousReferencePair` – présentée en PROG. 15.10 p175 – qui hérite de `AsynchronousReference` et qui permet de considérer une paire de références comme une référence grâce aux relations de typage du langage Java. Les deux méthodes `getHead()` et `getTail()` permettent de récupérer la référence asynchrone de tête ou de queue de la paire respectivement.

```
1 public interface AsynchronousReferencePair extends AsynchronousReference{
2     AsynchronousReference getHead();
3     AsynchronousReference getTail();
4 }
```

Prog. 15.10 – L'interface `AsynchronousReferencePair`.

Remarquons enfin que les paires *ne conservent plus* la propriété d'égalité définie en §14.2 p148. En effet, en reprenant la définition, on a $\mathtt{r}_1 = \mathtt{R}_{\mathtt{T}_1}(\mathtt{i})$ et $\mathtt{p} \equiv \mathtt{R}(\mathtt{i})$ alors que fatalement $\mathtt{r}_1 \neq \mathtt{p}$. Conscient de ce problème, nous cherchons des solutions. Nous envisageons deux solutions radicalement différentes qui sont :

- de ne plus conserver la relation d'égalité dans la définition de nos références ;
- de ne plus considérer les paires comme des références, mais comme des entités *faiblement équivalentes*.

La première solution rejoindrait la sémantique associée à la notion de référence du paquetage `java.lang.ref` mais elle nécessite des modifications dans un grand nombre de classes. La deuxième solution permet de garder une bonne partie de l'implémentation existante mais exige la définition d'une relation d'*équivalence faible* pour les paires de références dans laquelle le comportement des appels de méthodes est identique à celui des références, mais où la relation d'égalité n'est plus respectée : deux paires de références sont égales si les références qui les constituent sont égales :

$$< \mathtt{r}_2, \mathtt{r}_1 > (\mathtt{i}) ==< \mathtt{r}'_2, \mathtt{r}'_1 > (\mathtt{i}) \Leftrightarrow (\mathtt{r}_2 == \mathtt{r}'_2) \ \mathtt{et} \ (\mathtt{r}_1 == \mathtt{r}'_1)$$

On montre par récursion que cette relation est conservée par le chaînage multiple :

$$< \mathtt{r}_\mathtt{n}, \ldots, \mathtt{r}_1 > (\mathtt{i}) ==< \mathtt{r}'_\mathtt{n}, \ldots, \mathtt{r}'_1 > (\mathtt{i}) \Leftrightarrow (\mathtt{r}_\mathtt{n} == \mathtt{r}'_\mathtt{n}) \ \mathtt{et} \ \ldots \ \mathtt{et} \ (\mathtt{r}_1 == \mathtt{r}'_1)$$

15.3.2 Implémentation

L'implémentation de notre interface `AsynchronousReferencePair` mérite d'être détaillée. Essentiellement, notre classe devra masquer le chaînage des appels à la méthode `call()` (*cf.* §13.2.2 p143). En particulier, l'objet *future* (*cf.* §15.1.1.1 p159) renvoyé devra se comporter comme si l'appel était réalisé sur une référence asynchrone standard. Notre implémentation nommée `AsynchronousReferencePairImpl` est présentée en Prog. 15.11 p176. Notons l'utilisation de la constante `CallReflection.callMethod` qui est la réification de la méthode `AsynchronousReference.call()` pour éviter le coût d'un appel à `getMethod()` à chaque invocation.

Remarquons surtout qu'il est nécessaire d'implémenter une classe `FuturePair` pour simuler un *future* classique. C'est cette classe qui jongle avec les deux objets

```
1  public class AsynchronousReferencePairImpl
2              implements AsynchronousReferencePair {
3
4      // Can't be 'final' because of 'readObject()' !
5      protected AsynchronousReference head;
6
7      // **** Implementation of AsynchronousReference ****
8      public FutureClient call(final MethodOp method,
9                               final Object[] args,
10                              final Callback callback) {
11
12         final FutureClient headFuture =
13             head.call(CallReflection.callMethod,
14                       new Object[]{method, args});
15
16         return new FuturePair(this,
17                               method,
18                               args,
19                               Thread.currentThread(),
20                               callback,
21                               headFuture);
22     }
23
24     public Object getObject() {
25         return getTail().getObject();
26     }
27
28     /**
29        <p>Singleton implementation.</p>
30     */
31 ...
32 }
```

PROG. 15.11 – La classe `AsynchronousReferencePairImpl` implémente l'interface `AsynchronousReferencePair`.

futures retournés par chacun des deux appels. Par exemple lorsque le client invoque la méthode `waitForResult()` sur un *future* retourné par une paire de références, l'implémentation doit effectuer les opérations suivantes :

1. attendre que l'objet *future* lié au deuxième appel soit arrivé (si c'est déjà le cas, on n'attend évidemment pas) ;
2. récupérer le résultat du deuxième appel (c'est celui qui intéresse le client) ;
3. vérifier qu'une exception n'a pas été levée, si c'est le cas, l'exception est levée à nouveau ;
4. retourner le résultat au client.

Évidemment, la première étape implique la notification de la thread appelante de l'arrivée du *future* et donc des détails d'implémentations qui ne seront pas présentés ici[12] (verrou, thread d'attente du second *future*, gestion des exceptions liées au premier appel, gestion de délais (*timeout*), etc.).

15.3.3 Sémantiques chaînées

Nous avons évoqué dans l'exemple 15.3.1 les différences d'impact des sémantiques asynchrones lorsqu'elles sont associées à des références chaînées. En reprenant cet exemple, dans lequel on a :

$$\texttt{t = AR}_{\text{ThreadPooled}}\texttt{(SOR(AR}_{\text{FIFO}}\texttt{(i)))}$$

la sémantique non-concurrente est indispensable du point de vue du *stored object* `i` puisqu'il est non *thread safe*. Par contre, la sémantique concurrente permet aux clients (aux threads appelantes) de bénéficier d'un asynchronisme *complet*.

L'ordre de chaînage a donc une importance qui est à prendre en compte. Par exemple, en inversant les politiques de l'exemple précédent :

$$\texttt{t = AR}_{\text{FIFO}}\texttt{(SOR(AR}_{\text{ThreadPooled}}\texttt{(i)))}$$

le positionnement d'une politique FIFO par la thread cliente n'aura aucune influence sur la sémantique des appels de méthodes asynchrones : ils seront de toute façon concurrents. Il y a en effet trois chaînages d'appels par invocation sur `t` : le premier est lié à l'invocation de la méthode `call()` avec une politique FIFO. Le deuxième est l'invocation (éventuellement à distance) de la méthode `call()` du conteneur actif. La politique utilisée n'est pas définie ici. Le troisième appel est l'invocation de la méthode `call()` avec une politique utilisant un *threads pool*. Cette dernière politique

[12]Le lecteur intéressé peut se référer à la classe interne `FuturePair` contenue dans le fichier `src/mandala/rami/impl/AsynchronousReferencePairImpl.java` de la distribution de Mandala [214].

permettra à certains appels[13] d'être exécutés de manière concurrente. Ainsi, on peut distinguer deux catégories d'utilisation des politiques asynchrones :

- **politique serveur :** elle garantit la sémantique asynchrone des méthodes d'un objet indépendamment de la politique client ;
- **politique client :** elle influe sur l'efficacité des appels du client (latence), mais pas sur la sémantique asynchrone des appels.

Par conséquent, c'est au créateur de la référence asynchrone de bien choisir la politique qu'il souhaite utiliser. En outre, cette politique ne doit pas être changeable par un tiers. On peut en effet supposer que si une politique FIFO est associée à une référence asynchrone, c'est que c'est la mieux adaptée à l'objet référencé. Dans le cas où le client souhaite absolument utiliser une autre politique, il doit instancier lui même une référence asynchrone avec une copie de l'objet initial : en vertu de la propriété d'égalité vue en §14.2 p148 on ne peut en effet avoir deux références distinctes sur le même objet (dans la même machine virtuelle évidemment !).

15.4 Bilan

Nous avons présenté notre implémentation du concept de référence asynchrone en Java. Les interfaces ont été données et leurs implémentations étudiées. Notre problème initial a trouvé une solution par l'intermédiaire des paires de références. Si l'utilisation de la réflexion apporte le *dynamisme* recherché, il est néanmoins responsable de la lourdeur syntaxique des invocations qui ne ressemble guère à la notation standard des appels de méthodes. Il est donc temps de rendre plus *transparentes* nos invocations de méthodes asynchrones et éventuellement distantes. Le chapitre suivant est dédié à cette problématique.

[13] Rappelons que le nombre de threads est nécessairement borné.

Chapitre 16

Transparence

Les conteneurs actifs et les références asynchrones ont une caractéristique commune : le *dynamisme*. N'importe quel objet peut être accédé à distance – en devenant un *stored object* – et de manière asynchrone par le biais d'une référence asynchrone. Cette caractéristique fondamentale est rendue possible par l'utilisation du mécanisme de réflexion qui est standard en Java : les méthodes `call()` de nos conteneurs et de nos références asynchrones utilisent une méthode réifiée pour les invocations. L'avantage de ce mécanisme est d'être dynamique. mais il possède deux sérieux inconvénients :

- l'utilisation de la réflexion est relativement pénible à l'usage : pour chaque invocation, le développeur doit en effet "préparer son appel" : réifier la méthode, et placer les arguments dans un tableau ;
- le mécanisme réflexif affaiblit le typage puisque la signature de la méthode et les types des paramètres ne peuvent être vérifiés qu'à l'exécution.

Une solution consiste à déclarer une interface et à fournir au client un *proxy* qui implémente cette interface : lorsque le client invoque une méthode de ce *proxy*, l'invocation de méthode réflexif est réalisé automatiquement. Le *proxy* peut être généré par un compilateur spécialisé assurant ainsi le *typage fort*. C'est ainsi que fonctionne RMI (depuis le JDK v1.2 qui utilise la réflexion) et CORBA pour rendre *transparents* des appels de méthodes *distants*.

Cependant, on peut se demander quel intérêt apporte la réflexion si un compilateur doit être utilisé pour garantir le typage fort, il semble y avoir un paradoxe entre dynamisme et typage fort ! Nous allons voir qu'il n'en est rien. Mais avant, nous allons présenter le mécanisme généralement recherché : la *transparence totale*.

16.1 La transparence totale

Nous allons dans un premier temps définir ce que nous appelons *transparence totale* :

Définition 16.1.1 (Transparence totale)
Un appel de méthode asynchrone (et éventuellement distants) est totalement transparent *s'il est syntaxiquement indiscernable d'un appel de méthode standard (local).*

Pour qu'un appel de méthode asynchrone soit totalement transparent, il faut deux mécanismes :

1. un mécanisme qui rend l'appel effectivement asynchrone, on dira alors que le client utilise un *proxy asynchrone* ;
2. un mécanisme qui permette de récupérer le résultat de l'appel de façon transparente, on parlera de *futur transparent*.

Les proxy asynchrones peuvent utiliser n'importe lequel des concepts que nous avons étudiés pour l'expression de la concurrence §12.4.1 p134 dans les applications orientées objets.

Les futurs transparents peuvent utiliser le mécanisme d'*attente par nécessité* que nous avons présenté en §12.4.1.1 p134 : lorsqu'un client effectue un appel de méthode asynchrone, un objet *futur*, sous-type du type retourné par la méthode d'origine est retourné immédiatement. Lorsque le client utilise ce résultat, il est bloqué si le résultat effectif n'est pas encore disponible.

Nous allons présenter deux implémentations permettant d'assurer une totale transparence des appels de méthodes asynchrones (et éventuellement distant).

16.1.1 Héritage

Dans ProActive (*cf.* §9.3 p82) le concept utilisé pour exprimer la concurrence est l'*objet actif*. Un proxy asynchrone est une instance de classe qui hérite de la classe de l'objet original. Un futur transparent est une instance de classe qui hérite de la classe retournée dans la méthode originale. Par exemple, après avoir rendu un objet actif, `a`, instance de classe `A`, le client utilise un *proxy* `p` dont la classe `P` hérite de `A`. Toutes les méthodes de `A` sont redéfinies dans `P`. Par exemple, s'il existe une méthode dans `A` déclarée par : `T m(B b, C c)`, alors dans la classe proxy `P`, la méthode `m` retourne un futur transparent dont le type est une sous-classe de `T`. C'est cette sous-classe qui implémente le mécanisme d'*attente par nécessité* : chaque appel donne lieu à un test pour vérifier si le résultat est disponible. Si le test est vrai, la méthode de la classe mère est invoquée, sinon, le client est bloqué. Lorsque le résultat devient disponible, toutes les threads en attente sont réveillées.

Cette solution possède quelques problèmes que nous avons déjà mentionnés dans notre présentation de ProActive et que nous reprenons en partie ici :

- les classes déclarées `final` ne peuvent pas être héritées, ce qui empêche d'une part leurs instances de devenir actives et d'autre part la création d'objets *futurs* de ce type (problème n°3 p83) ;
- les méthodes déclarées `final` ne peuvent être redéfinies, ce qui empêche les objets futurs de gérer le résultat des appels asynchrones (problème n°4 p83) ;
- les méthodes qui retournent un type primitif ne peuvent être surchargées pour retourner un objet *futur*, elles sont donc nécessairement synchrones (problème n°5 p83) ;
- les objets qui accèdent directement aux champs publics des objets futurs ne peuvent être bloqués en attente du résultat (problème n°6 p83).
- le problème habituel lié à la sémantique du mot-clé `static` dans un cadre distribué n'est pas résolu : si un champ déclaré `static` est modifié, la nouvelle valeur devrait être répercutée sur toute les machines virtuelles qui ont chargé la classe (problème n°7 p84) ;
- les objets qui accèdent directement aux champs `public` d'un objet actif sont problématiques : si l'objet actif est distant, le champ auquel on accède effectivement est celui du proxy et non celui de l'objet actif, d'où un problème de cohérence (problème n°8 p84);

Les deux derniers problèmes méritent une explication supplémentaire. En effet, accéder à un champ d'instance ou de classe public ne pose guère de problème (ni pour l'aspect concurrent, ni pour l'aspect distribué) s'il est constant *i.e.* déclaré `final`[1]. Or, 99,9% des champs de classes dans le JDK v1.4[2] sont déclarés `final`. On pourrait donc penser que le problème d'accés aux champs est mineur. Cependant, 20% seulement des champs d'instances sont constants et le problème reste donc entier : on trouve en effet 8 champs d'instances publics non constants pour 100 classes publiques.

Pour éviter l'ensemble de ces problèmes, nous proposons une implémentation à base d'interfaces.

16.1.2 Interface

Les interfaces en Java ne possèdent pas de champ (ou alors ils sont constants) et leurs méthodes sont implicitement déclarées `public`. En outre, elles ne peuvent être `final`. Aussi, en utilisant exclusivement des interfaces pour la transparence asynchrone, on évite ainsi tous les problèmes précédemment évoqués.

[1] A ne pas confondre avec les classes *immutables* telle que `java.lang.String`.

[2] Seules les classes dont le nom complet est préfixé par `java` ont été considérées.

Java propose depuis le JDK v1.3 les *proxy dynamiques* [146] : la classe `java.lang.reflect.Proxy` permet de générer dynamiquement des classes qui implémentent un certain nombre d'interfaces spécifiées à leur création[3]. Le code métier du proxy est une instance de classe qui implémente l'interface `java.lang.reflect.InvocationHandler`. Les proxy dynamiques permettent donc de réaliser une solution totalement transparente à base d'interfaces.

Nous proposons une implémentation de l'interface `java.lang.reflect.InvocationHandler` dans la classe `AsynchronousProxyInvocationHandler`, présentée en PROG. 16.1 p182 qui redirige les appels vers une référence asynchrone associée à la construction. Les proxy générés par l'intermédiaire de cette classe sont des *proxy asynchrones*.

```
1  public class AsynchronousProxyInvocationHandler
2      implements InvocationHandler, Serializable {
3
4      protected final AsynchronousReference asynchronousReference;
5      /**** InvocationHandler implementation ****/
6      public Object invoke(final Object proxy,
7                           final Method method,
8                           final Object[] args) throws Throwable {
9          final Class returnType = method.getReturnType();
10         final Class[] resultInterfaces;
11         if (returnType.isInterface()) {
12             resultInterfaces = new Class[] {returnType};
13         }else{
14             resultInterfaces = returnType.getInterfaces();
15         }
16         final FutureClient future = asynchronousReference.call(method, args);
17         return Proxy.newProxyInstance(resultInterfaces,
18                                       new FutureProxy(future));
19     }
20     /* Singleton implementation. */
21 }
```

PROG. 16.1 – La classe `AsynchronousProxyInvocationHandler` qui représente les *proxy asynchrones totalement transparents*.

Comme on le voit, le résultat de chaque appel de méthode de notre proxy asynchrone est remplacé par une instance de la classe `FutureProxy` – classe interne présentée en PROG. 16.2 p183. Une instance de cette classe est un *futur transparent* qui encapsule l'objet `FutureClient` renvoyé par l'appel asynchrone. Lorsqu'une invocation de méthode est réalisée sur ce futur, il attend la disponibilité du résultat

[3]La machine virtuelle a du être modifiée à cet effet.

réel en utilisant la méthode `waitForResult()` et redirige l'appel sur le résultat effectif renvoyé ensuite.

```
1  class FutureProxy implements InvocationHandler, Serializable {
2      final FutureClient future;
3
4      FutureProxy(final FutureClient future) {
5          this.future = future;
6      }
7      /**** InvocationHandler implementation ****/
8      public Object invoke(final Object proxy,
9                           final Method method,
10                          final Object[] args) throws Throwable {
11         // Object's method must not be redefined.
12         if (method.getDeclaringClass().equals(Object.class)) {
13             return method.invoke(this, args);
14         }
15         final Object result = future.waitForResult();
16         return method.invoke(result, args);
17     }
18 }
```

Prog. 16.2 – Implémentation des *futurs transparents*.

La figure 16.1 illustre ce mécanisme : un client effectue un appel de méthode, `p.m()`, sur un proxy asynchrone totalement transparent (1). Ce dernier utilise une référence asynchrone, `ar`, et encapsule l'appel original en un appel à la méthode `ar.call()` (2) rendant de fait l'appel asynchrone (4'). La méthode `call()` créée un `FutureClient` (3), `fc`, qui est encapsulé dans un *proxy résultat*, `Proxy(r)`, assurant ainsi le typage fort du retour de la méthode. Le client peut donc utiliser le résultat, `r`, comme d'habitude. Si un appel de méthode, `r.foo()` est effectué sur ce dernier (4), le *proxy résultat* utilise le `FutureClient` pour assurer que l'appelant reste bloqué si l'appel orginal n'est pas terminé en invoquant la méthode `fc.waitForResult()` (5). Une fois le résultat réel disponible (5'), l'appel original `foo()` lui est transmis (6).

Nous avons déjà rencontré un exemple de code utilisant la transparence totale en §2.2.5 p20.

16.1.3 Bilan

L'avantage de la transparence totale n'est pas tant lié à la syntaxe qu'elle autorise mais plus à la compatibilité qu'elle permet avec des codes non prévus pour être exploités dans un cadre asynchrone (et éventuellement distant). Cette faculté est liée au typage : l'utilisation de sous-types (par héritage ou par interface) permet de

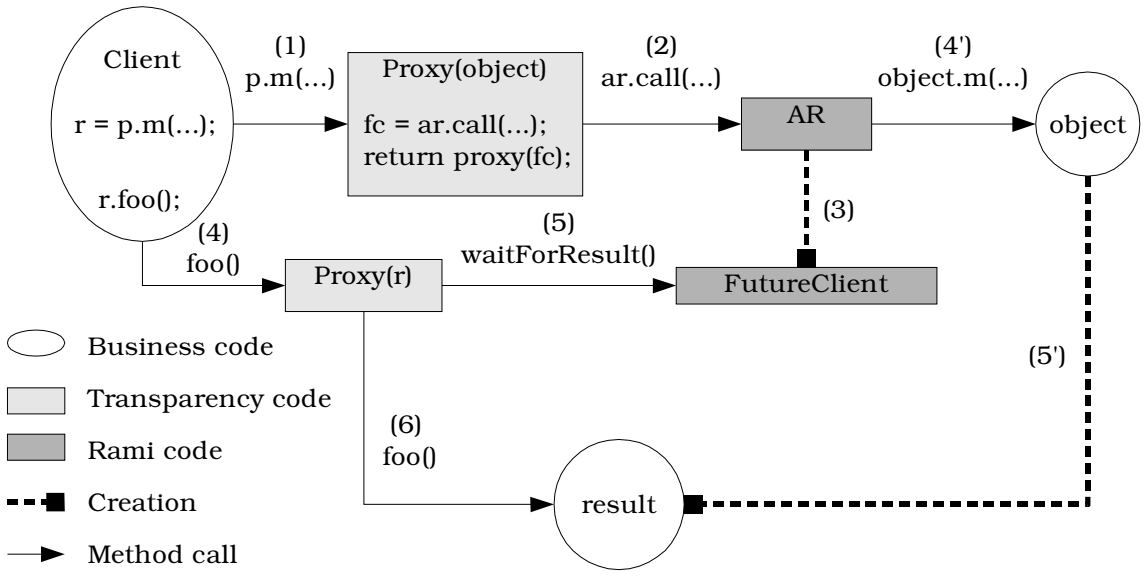


FIG. 16.1 – Appel de méthode sur un proxy asynchrone totalement transparent.

passer un proxy asynchrone totalement transparent à une méthode qui attendait un objet standard.

La solution à base d'interfaces, si elle évite les problèmes associés à l'utilisation de l'héritage contraint très fortement son cadre d'utilisation : pour obtenir un proxy asynchrone totalement transparent sur un objet, il faut :

- que la classe de l'objet implémente au moins une interface ;
- que ce soient justement les méthodes de ces interfaces qui doivent être accédées de manière asynchrone (et éventuellement distante) ;
- que le type de retour des méthodes en question soit une interface ou implémente une interface.

Dans le JDK v1.4, il y a 3 950 classes[4] dont 57% sont publiques. Parmi celles-ci, 20 % sont des interfaces et 30% implémentent une interface. Seulement 5% des méthodes publiques sont déclarées par des interfaces. Peu d'entre-elles retournent une interface ou un type qui en implémente une : sur les 17 254 méthodes publiques que comptent le JDK v1.4, seules 653 (4%) entrent dans le critère précédent et sont donc exploitables de manière asynchrone en utilisant un proxy totalement transparent. Par conséquent, il semble clair que notre mécanisme permet rarement l'utilisation de tels proxy dans une application qui n'a pas été prévue pour cela au départ.

[4]Seules les classes dont le nom complet est préfixé par `java` ont été considérées.

16.1.4 Problèmes inhérents à la transparence totale

Nous allons soulever les problèmes qui sont intrinsèquement liés à la transparence totale.

16.1.4.1 Gestion des exceptions

Nous avons déjà évoqué le problème lié à la gestion des exceptions dans un appel asynchrone totalement transparent en §12.4.1.1 p136. Nous n'y reviendrons pas. Notons simplement que ce problème est aussi présent dans notre implémentation à base d'interfaces.

16.1.4.2 Conscience du développeur

Le problème majeur de la transparence totale est justement lié au fait qu'elle est totale : elle permet au développeur d'oublier l'aspect asynchrone et éventuellement distant de ses appels.

Dans un premier temps, une séparation doit-être effectuée entre un appel local (avec une sémantique "passage par référence") et un appel distant (sémantique "passage par sérialisation"). Dans notre plate-forme, l'aspect distant d'une référence asynchrone est masqué par le fait que ce n'est pas tant l'objet qui est distant, mais le conteneur dans lequel il est. Nous avons vu que dans JACOb, les *types primitifs distants* sont clairement identifiés par le type `Remote` (*cf.* §10.2.3.3 p102). Le problème c'est que cette particularité n'apparaît pas dans le type d'un proxy asynchrone totalement transparent.

Pour l'aspect asynchrone, la situation est pire. L'utilisation du paradigme d'appel de méthodes asynchrones permet d'exprimer la concurrence d'exécution au sein d'une application. On peut supposer que cela mène à de meilleures performances, d'autant que si les appels sont distants, ils permettent d'exploiter le parallélisme. Cependant, pour être efficace, un développeur qui utilise des appels asynchrones doit suivre la règle suivante :

Définition 16.1.2 (Règle de développement concurrent efficace)
On obtient une efficacité maximum dans un paradigme à base d'appels de méthodes asynchrones lorsque :

- *les appels de méthodes sont réalisés le plus tôt possible ;*
- *les résultats sont utilisés le plus tard possible.*

Cela peut sembler évident, mais ce n'est pas la manière naturelle de développer une application séquentielle. Considérons le code suivant :

```
1 // Step 1
2 MyClass myObject = new MyClass();
```

```
3 // Step 2
4 MyInfos infos = myObject.myMethod(myParameters).getInfos();
5 // Step 3
6 doSomething();
```

Ce schéma de codage est très courant en Java. Si la variable `myObject` référence en fait un proxy asynchrone totalement transparent (en remplaçant `new MyClass()`), le code résultant ne sera pas plus efficace pour autant. Il sera même probablement plus lent : la gestion de la concurrence a un coût. Pour être efficace, le code devrait être écrit de la manière suivante :

```
1 // Step 1
2 MyClass myObject = new MyClass();
3 // Step 2
4 MyResult result = myObject.myMethod(myParameters);
5 // Step 3
6 doingSomething();
7 // Step 4
8 MyInfos infos = result.getInfos();
```

Ceci n'est manifestement pas un code séquentiel habituel en Java.

Il n'est donc pas raisonnable d'affirmer que la transparence totale permet de rendre concurrente (et éventuellement parallèle) une application séquentielle sans modifications majeures. Ces modifications peuvent être réalisées par un outil (automatique, aide à la décision, etc.), mais dans ce cas pourquoi utiliser la transparence totale ?

Aussi, nous pensons que le développeur doit-être conscient de l'asynchronisme de son appel pour écrire un code concurrent efficace. Nous proposons à cet effet un mécanisme à mi-chemin entre la transparence totale et l'utilisation explicite de la méthode `call()` : la semi-transparence.

16.2 La semi-transparence

Le mécanisme d'appel de méthodes asynchrones, éventuellement distants, *semi-transparent* que nous proposons permet de résoudre les problèmes de la *transparence totale* étudiée précédemment.

16.2.1 Syntaxe

Après notre étude de la transparence totale, nous sommes arrivés à la conclusion suivante :

Un appel de méthode asynchrone ne doit pas avoir la même syntaxe qu'un appel synchrone.

Pour autant, le typage fort doit-être assuré (sinon, la méthode `call()` pourrait-être utilisée directement !). Nous proposons donc la syntaxe suivante :

Notation 16.2.1 (Syntaxe d'un appel asynchrone semi-transparent)
Si une méthode publique *possède la signature suivante :*

```
T m(A1 a1, ..., An an)
```

alors, la version asynchrone et semi-transparente possède la signature :

```
FutureClient rami_m(A1 a1, ..., An an, Callback c)
```

En outre, si la méthode `m` *déclare lever des exceptions avec la clause* `throws`*, elles ne sont pas déclarées dans* `rami_m()`[5].

Cette syntaxe résout plusieurs problèmes :

- le développeur connaît la nature asynchrone des appels effectués de par la signature de la méthode qui n'est pas la même que celle d'origine ;
- les exceptions sont gérées à la récupération du résultat de manière active via le *futur* (*cf.* §14.4.1 p152) ou de manière passive en utilisant le *callback* (*cf.* §14.4.2 p152) ;
- le typage fort est assuré lors du passage des arguments par l'utilisation des mêmes types de paramètres que la méthode d'origine.

Notons que le typage fort n'est pas assuré à la récupération du résultat (active ou passive), puisque un transtypage (*cast*) entre le type générique `Object` retourné par les méthodes de l'interface `MethodResult`[6] (*cf.* §15.1.1.1 p159) et le type de résultat souhaité doit être effectué. Ce problème est mineur dans la mesure où cette pratique est commune en Java (le paquetage `Collection` en est un bon exemple) et surtout où les *types génériques* du JDK v1.5 apporteront une solution immédiate : il sera possible en effet de déclarer `FutureClient<T>` comme type de retour de la méthode `rami_m()`.

Nous allons voir en §16.2.3 p189 que les versions asynchrones et semi-transparentes des méthodes des objets peuvent être générées automatiquement. Mais avant, nous devons définir ce que nous appelerons la *vue* d'un objet.

[5]Le préfixe `rami_` n'est pas celui que nous préférons. Le symbole `#` était prévu à l'origine mais il ne fait pas partie des caractères autorisés pour les identificateurs dans la grammaire du langage Java.

[6]Rapellons que celle-ci contient l'ensemble des méthodes relatives à la récupération du résultat : si elle est active, alors c'est `FutureClient`, un sous-type, qui doit-être utilisé ; si elle est passive, c'est dans les paramètres de la méthode `Callback.done()` que l'on retrouve le type `MethodResult`. On se réfèrera aux sections §15.1.1 p158 et §15.1.2 p160 pour de plus amples détails.

16.2.2 Point de vue du client d'un objet

Nous distinguons l'objet qui est référencé par une référence asynchrone de la *vue* qu'en a un client. Par exemple, deux clients peuvent avoir des vues distinctes d'un même objet. C'est la notion d'interface : un objet peut implémenter plusieurs interfaces, lesquelles représentent des vues distinctes de lui. Pour ne pas mélanger le concept d'interface au sens Java et les interfaces au sens général, ces dernières serons réellement appelées des *vues*.

Aussi, nous considérons qu'un objet implémente toujours un ensemble de vues : toute combinaison de ses méthodes publiques représente une vue particulière de l'objet.

Par exemple, considérons la classe `java.lang.Object`. La liste de ses méthodes publiques est donnée par l'instruction *shell* suivante :

```
> javap -public java.lang.Object
public class java.lang.Object{
    public native int hashCode();
    public java.lang.Object();
    public final native void notify();
    public final native void notifyAll();
    public final void wait() throws java.lang.InterruptedException
    public final native void wait(long) throws java.lang.InterruptedException
    public final void wait(long,int) throws java.lang.InterruptedException
    public final native java.lang.Class getClass();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
}
```

On peut aussi considérer que cette classe implémente les vues arbitraires suivantes (déclarées sous forme d'interface Java) :

```
public interface Anonymous1{
    public native int hashCode();
    public final native java.lang.Class getClass();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
}

public interface Anonymous2{
    public java.lang.String toString();
}
```

même si la classe `java.lang.Object` ne le déclare pas explicitement.

Pour communiquer avec un objet, un client doit connaître des méthodes que l'objet implémente, *i.e.* doit en avoir une vue. Dans la suite, nous ne considérerons

que l'aspect asynchrone et nous parlerons de *vue* pour désigner une *vue asynchrone* c'est à dire une vue qui offre les moyens d'utiliser un objet de manière asynchrone.

16.2.3 Le compilateur `jayac`

Le compilateur `jayac` est un *générateur de vues asynchrones*. Pour une classe donnée, il génère une *vue asynchrone* qui contient l'ensemble de ses méthodes publiques déclarées de manière asynchrone avec la syntaxe définie précédemment. Cette vue est une classe Java dont le nom est celui de la classe d'origine préfixé par le nom de paquetage `jaya`. Une instance de cette classe est un *proxy asynchrone semi transparent*.

Les super-classes et les interfaces implémentées par la classe spécifiée subissent le même traitement de manière récursive générant ainsi une hiérarchie de vues (de classes) symétrique de celle des classes d'origines.

Prenons pour exemple, la classe `java.io.FileWriter` :

```
1  > javap -public java.io.FileWriter
2  public class java.io.FileWriter extends java.io.OutputStreamWriter{
3      public java.io.FileWriter(java.io.File) throws java.io.IOException;
4  ...
5      public java.io.FileWriter(java.io.FileDescriptor);
6  ...
7  }
```

Cette classe ne définit aucune méthode. Elle hérite simplement de la classe `java.io.OutputStreamWriter`. Aussi, notre générateur de vues produira simplement les constructeurs nécessaires dans la classe `jaya.java.io.FileWriter` :

```
1  > javap -public jaya.java.io.FileWriter
2  public class jaya.java.io.FileWriter extends jaya.java.io.OutputStreamWriter{
3  ...
4     public jaya.java.io.FileWriter(java.io.File) throws java.io.IOException;
5     public jaya.java.io.FileWriter(java.io.File,
6                                    Framework.Factory) throws java.io.IOException;
7     public jaya.java.io.FileWriter(java.io.FileDescriptor);
8     public jaya.java.io.FileWriter(java.io.FileDescriptor,
9                                    Framework.Factory);
10 ...
11    public static jaya.java.lang.Object getInstance(AsynchronousReference);
```

Chaque constructeur est surchargé pour prendre en argument un *factory*. Ce dernier permet de paramétrer le type de références asynchrones qui seront créées pour référencer le nouvel objet (*stored object reference* ou paire de références par exemple).

Les constructeurs sans cet argument utilisent un *factory* par défaut. Ainsi une instruction `new` standard permet de créer un objet (éventuellement à distance) grâce à la méthode `newInstance()` du *factory* et d'en obtenir un *proxy* asynchrone semi transparent.

Notons qu'il est aussi possible de créer un *proxy* asynchrone sur un objet déjà instancié si l'on en possède déjà une référence asynchrone en utilisant la méthode de classe `getInstance()` qui assure la caractéristique d'égalité énoncée en §14.2 p148 via le *singleton* vu en §15.2.1 p161.

Aucune méthode n'apparaît dans cette vue conformément à la classe d'origine.

La classe `java.io.FileWriter` héritant de `java.io.OutputStreamWriter`, la vue asynchrone de cette dernière a aussi été générée. Observons d'abord les méthodes de la classe d'origine (les constructeurs sont générés de manière identique à ceux de la vue précédente). Nous allons nous limiter à une méthode, le principe étant toujours le même :

```
1  > javap -public java.io.OutputStreamWriter
2  public class java.io.OutputStreamWriter extends java.io.Writer{
3  ...
4      public void write(char[],int,int) throws java.io.IOException;
5  ...
```

La vue correspondante est donnée ci-dessous :

```
1  > javap -public jaya.java.io.OutputStreamWriter
2  public class jaya.java.io.OutputStreamWriter extends jaya.java.io.Writer{
3  ...
4      public mandala.rami.FutureClient rami_write(char[],
5                                                  int,
6                                                  int);
7      public mandala.rami.FutureClient rami_write(char[],
8                                                  int,
9                                                  int,
10                                                 mandala.rami.Callback);
11     public void write(char[],int,int) throws java.io.IOException;
12 ...
13 }
```

On observe les méthodes asynchrones préfixées par `rami_` et retournant un `FutureClient`. Notons que ces méthodes ont la même signature que l'originale au préfixe et au type de retour près. Elles sont toutes surchargées par une méthode qui prend les mêmes arguments augmenté d'un objet de type `Callback`. Par ailleurs aucune ne lève une exception. A la récupération du résultat, l'appelant devra vérifier la nature de la terminaison de son appel.

Finalement, dans un souci de commodité, la version synchrone de l'appel est aussi proposée.

La classe `OutputStreamWriter` hérite de `Writer` dont la vue associée n'est pas présentée ici. Cette dernière classe hérite de `java.lang.Object`. De la même manière, tous les proxy asynchrones semi-transparents héritent de sa vue associée : `jaya.java.lang.Object`. Cette classe fournit des fonctionnalités communes à tous les proxy semi-transparents tels que : la référence asynchrone associée ou l'instance `SingletonGiver` qui garantit la propriété d'égalité. Nous ne présenterons pas cette vue ici. Par contre, à titre d'illustration, nous donnons l'implémentation de la méthode `write()` de la vue `jaya.java.io.OutputStreamWriter` :

```
1  public FutureClient rami_write(char[] charValues0,
2                                 int intValue1,
3                                 int intValue2,
4                                 Callback callback) {
5
6      Object args[] = new Object[]{charValues0,
7                                   new java.lang.Integer(intValue1),
8                                   new java.lang.Integer(intValue2)};
9
10     return asynchronousReference.call(writeMethod, args, callback);
11 }
```

Dans cet extrait, on suppose que l'on dispose d'un champ static `writeMethod`[7] qui est la réification de la méthode synchrone associée. Les champs de types primitifs sont encapsulés dans des classes englobantes.

Enfin, la version synchrone a exactement la même signature que la méthode originale. Le code est le suivant :

```
1  public void write(char[] charValues0,
2                    int intValue1,
3                    int intValue2) throws java.io.IOException{
4      try{
5          rami_write(charValues0,
6                     intValue1,
7                     intValue2).waitUntilResultAvailable();
8      }catch (java.lang.Throwable e) {
9          if (e instanceof java.io.IOException) throw (java.io.IOException) e;
10         throw new RuntimeException("Undeclared exception thrown", e);
11     }
12 }
```

[7]Évidemment, un algorithme de résolution des conflits de noms doit être trouvé pour éviter des champs homonymes dans le cas de surcharge de méthode. Cet algorithme ne sera pas présenté.

Les versions synchrones utilisent la référence asynchrone associée au *proxy* pour acheminer l'appel de méthode vers l'objet métier. Ce dernier pouvant être distant (par l'intermédiaire d'une référence sur un *stored object* distant), il n'est de toute façon pas possible d'utiliser une référence locale Java standard[8].

Un traitement particulier doit être effectué pour la levée des exceptions : les méthodes `waitUntilResultAvailable()` (utilisée pour les méthodes procédurales) et `waitForResult()` (utilisée pour les méthodes fonctionnelles) de la classe `FutureClient` peuvent lever l'exception générique `Throwable`. Il est donc nécessaire de vérifier que si une exception survient, elle est bien du type déclaré par la version synchrone, sinon, une exception non-contrôlée doit être levée.

En introduction, nous avons évoqué le paradoxe qui semble exister entre le dynamisme apporté par la réflexion et le typage fort : si, clairement, la semi-transparence garantit un typage fort, diminue-t'elle le dynamisme que nous recherchons ? On pourrait le penser. En effet, pour obtenir un proxy asynchrone semi transparent, une phase de compilation est nécessaire pour générer la vue associé à une classe. Cependant, cette phase ne requiert pas le code source de la classe d'origine. En utilisant à nouveau la réflexion, les vues peuvent être générées directement à partir de l'API standard sans même avoir besoin d'analyser le *bytecode* Java de la classe d'origine. Dans l'exemple précédent, nous avons généré la vue asynchrone de la classe `java.io.FileWriter`. A l'instanciation, d'une telle vue, nous obtenons un proxy asynchrone semi-transparent d'une instance de la classe `java.io.FileWriter`. C'est cette instance qui contient le code métier proprement dit. Aussi, le dynamisme reste bien conservé. Par contre, la génération de code à la volée n'est pas possible ici, puisque l'utilisation d'un proxy asynchrone semi-transparent nécessite d'en connaître le type.

Notons toutefois une limitation de l'utilisation de la réflexion pour la génération des vues : les références récursives directes ou indirectes entre une classe et sa vue associée sont problématiques. Un exemple de récursion indirecte nous servira d'illustration : supposons qu'une classe `A` utilise la vue `jaya.B` et que la classe `B` associée utilise des instances de `A`. Le fichier `A.java` ne peut compiler sans l'existence de `jaya.B.class`. Or, pour que `jaya.B.java` soit généré, il faut que `B.class` le soit aussi ce qui requiert la compilation du fichier `A.java` ! Une solution à ce problème est de créér une classe factice `B` contenant seulement des méthodes vides, sans référence à la classe `A`. La compilation de cette classe permet au compilateur d'en générer la vue `jaya.B`. Le remplacement de la classe factice `B` par la classe métier contenant

[8]La sémantique serait en outre différente selon la nature de l'appel : la version synchrone utiliserait un passage par référence, la version asynchrone un passage par copie. Cela n'est pas acceptable.

des références à `A` permettra à l'ensemble de compiler. Une mise en œuvre automatisée de cette solution utilisant l'utilitaire Ant [72] est disponible dans le répertoire `Examples` de l'arborescence de Mandala [214]. Il serait évidemment préférable de réaliser un générateur de vues à partir du code source des classes Java.

16.3 Bilan

Le mécanisme de semi-transparence offre des caractéristiques très intéressantes du point de vue du développement d'applications concurrentes. Il offre en effet un bon compromis entre le mécanisme réflexif à base d'appel à la méthode `call()` qui, outre la lourdeur syntaxique, ne garantit pas le typage fort et la transparence totale qui possède l'inconvénient majeur de *conscience du développeur* et dans une moindre mesure de gestion des exceptions délicate.

Les vues générées par notre outil ont une certaine symétrie par rapport à leur classe d'origine. Par exemple, une vue hérite nécessairement, directement ou indirectement de la vue `jaya.java.lang.Object`, le symétrique de la classe `java.lang.Object`. Cette symétrie facilite l'assimilation de notre approche : le développeur retrouve l'ensemble des concepts qui lui sont familiers : appels de méthodes et hiérarchie de classes.

L'inconvénient de cette solution est qu'elle n'est plus compatible avec les codes existants : une méthode qui prend en argument un objet de type `java.io.FileWriter` par exemple ne peut bénéficier d'une version asynchrone sans avoir été codé à cet effet (grâce à la propriété d'égalité énoncée en §14.2 p148, la méthode en question peut récupérer un proxy transparent sur l'objet qui lui est passé en paramètre et bénéficier ainsi d'une version asynchrone sans que cela apparaisse dans son interface). Cependant, cet inconvénient est à relativiser : si la méthode n'a pas été prévue pour utiliser une version asynchrone, alors lui fournir un proxy asynchrone totalement transparent n'améliorera vraisemblablement pas ses performances et risque même de les diminuer.

Cette incompatibilité des types a cependant deux avantages :

- elle force le développeur a prendre conscience de l'aspect asynchrone de son appel ;
- elle évite des ambiguités de noms entre les versions standards des classes et leurs vues associées.

Ce dernier point mérite une petite explication. On ne peut utiliser à la fois la classe `java.io.FileWriter` et sa vue associée sans être explicite en raison de l'homonymie des classes. Par exemple, dans le code suivant :

```
1 import java.io.*;
2 import jaya.java.io.*;
```

```
3  ...
4      Writer writer = new FileWriter("foo.txt");
5             writer = new FileWriter("bar.txt");
```

on ne peut savoir si la variable `writer` référence une instance de la classe `java.io.FileWriter` ou une instance de sa vue associée, un proxy asynchrone semi transparent de type `jaya.java.io.FileWriter`.

Si par abus de langage, on considère un proxy asynchrone comme une référence asynchrone[9], alors, on peut dire que dans l'exemple précédent, on ne peut savoir si la variable `writer` est une référence synchrone ou asynchrone. L'homonymie des classes et de leurs vues asynchrones contraint le développeur à utiliser des noms complets. Le code précédent, qui ne compile pas, peut être réécrit de plusieurs façons. Par exemple, si l'essentiel du code est synchrone, on préférera écrire :

```
1  import java.io.*; // Use shortcuts for synchronous class only
2  ...
3      Writer writer = new FileWriter("foo.txt");
4      jaya.java.io.Writer writerProxy = new jaya.java.io.FileWriter("bar.txt");
```

Si au contraire, l'essentiel du code est asynchrone, on utilisera :

```
1  import jaya.java.io.*; // Use shortcuts for asynchronous class only
2  ...
3  java.io.Writer writer = new java.io.FileWriter("foo.txt");
4     Writer writerProxy = new FileWriter("bar.txt");
```

[9]Il y a une injection entre l'ensemble des proxy asynchrones et l'ensemble des références asynchrones en raison de la relation d'égalité et du mécanisme de singleton qui le réalise : on ne peut avoir deux proxy asynchrones distincts sur la même référence asynchrone.

Chapitre 17

Conclusion

Nous avons exposé le problème auquel nous avons été confronté qui est, rappelons-le, lié à la programmation d'applications distribuées (*cf.* §10.2.4 p104) : l'expression de la concurrence dans les applications orientés objets. Nous avons présenté un état de l'art et proposé plusieurs solutions. Le concept de références asynchrones nous est apparu comme le plus prometteur. Nous l'avons défini et implémenté (en version locale et en version distante). Nous avons trouvé la solution à notre problème initial au travers des paires de références : la méthode `call()` de notre conteneur actif est asynchrone côté serveur. En chaînant des références asynchrones, on peut offrir un asynchronisme complet. Enfin, le typage fort qui manquait dans l'utilisation de la méthode `call()` a été réintroduit par l'utilisation des mécanismes de transparence. A ce titre, nous favorisons la semi-transparence qui nous semble être mieux adaptée au paradigme d'appels de méthodes asynchrones puisqu'il permet plus facilement au développeur de prendre conscience de la nature asynchrone de ses appels et donc d'écrire un code plus performant qui colle à la règle 16.1.2.

Il reste à effectuer des mesures de performance et à étudier des exemples de code pour vérifier l'adéquation de notre concept à la programmation orientée objets.

Cinquième partie

Plate-forme générale : Mandala

Chapitre 18

Présentation de la plate-forme Mandala

Dans ce chapitre, nous allons brièvement présenter la plate-forme Mandala [212], vérifier la pertinence de nos modèles pour l'expression de la concurrence et/ou de la distribution sur des exemples de codes et mesurer les performances de nos implémentations.

18.1 Relation entre les conteneurs actifs et les références asynchrones

Les deux concepts que sont les conteneurs actifs (*cf.* §III p75) et les références asynchrones (*cf.* §IV p123) sont présentés dans deux sous-*packages* du *package* central `mandala`. On retrouve donc `mandala.jacob` pour les classes relatives aux conteneurs actifs, et `mandala.rami` pour celles relatives aux références asynchrones. Notons aussi la présence d'une bibliothèque de classes utilitaires regroupées au sein du *package* `mandala.util` telles que `SingletonGiver` (*cf.* §15.2.1 p161), `ExceptionsHandler` (*cf.* §10.2.3.3 p102) ou `ThreadPool` (utilisée par la politique concurrente `ThreadPooledPolicy` présentée en §15.8 p168). Ces classes sont exploitées par les deux *packages* précédents.

Si Mandala fait appel à de nombreux concepts (conteneur actif, gestionnaire d'exception, *stored object*, référence asynchrone, réponse anticipée, annulation, *callback*, transparence, ...), un utilisateur de la plate-forme manipulera essentiellement des *proxys semi-transparents* (*cf.* §16.2 p186). Les deux concepts fondamentaux sont ainsi masqués ce qui simplifie grandement la tâche du développeur. Cependant, ils restent accessibles : il est possible de récupérer la référence asynchrone associée à un proxy asynchrone, et s'il s'agit d'une référence sur un *stored object*, de récupérer

la clé et le conteneur actif qui lui sont associés.

18.2 La classe centrale `Framework`

Nous avons vu en §16.2.3 p189 que les classes des proxy asynchrones semi-transparents sont générées par le programme `mandala.rami.transparency.semi.Jayac`. L'instanciation d'un tel proxy requiert la spécification d'un objet *factory* [81], qui créé les références asynchrones associées. Nous avons mentionné l'existence d'un *factory* par défaut. Cet objet est en réalité partagé dans toute la machine virtuelle au sein d'une classe centrale nommée `Framework`. La méthode `getFactory()` de celle-ci permet de récupérer le *factory* par défaut. Symétriquement, la méthode `setFactory()` permet de positionner le *factory* par défaut. Le *factory* par défaut est utilisé à la fois par les proxys asynchrones totalement transparents et par les proxy asynchrones semi-transparents.

Enfin, deux méthodes sont proposées dans la classe `Framework` : `getTotalTransparentAsynchronousProxy(Object)` et `getSemiTransparentAsynchronousProxy(Object)`. Elles permettent de récupérer le proxy asynchrone totalement (respectivement semi-) transparent associé à un objet, ou de le créer. Dans le cas des proxys asynchrones semi-transparents, le vrai type de l'objet spécifié doit être trouvé : il faut en effet instancier la classe dont le nom est celui du type réel de l'objet spécifié préfixé par `jaya.` ; les mécanismes réflexifs du langage Java sont utilisés à cet effet.

18.3 Mise en œuvre de méthodes de génie logiciel

La conception de la plate-forme a nécessité la mise en place d'un certain nombre d'outils qui facilitent le développement et une approche de type génie logiciel. Citons par exemple :

- le passage à Ant [72] pour la compilation au lieu de make [71], pour des raisons de portabilité ;
- le développement sous Eclipse [73] et sous le JDEE [112] d'Emacs [70] pour l'amélioration du code (*code refactoring*) ;
- l'utilisation de CVS [173, 213] pour le *versioning* ;
- et surtout, la découverte de la programmation extrême[1] [38] qui a directement influencé notre façon de programmer en mettant en œuvre de manière quasi[2]-systématique des tests unitaires à l'aide du *framework* JUnit [83].

[1] Merci à Georges Eyrolles qui m'a ouvert les yeux !

[2] Personne n'est parfait !

Nous allons détailler ce dernier point en prenant pour exemple le test unitaire d'un appel de méthode asynchrone semi-transparent. Le but de ce test est de vérifier que pour une méthode `m()` donnée :

- l'appel `rami_m()` ne bloque pas la *thread* courante ;
- la méthode `m()` est bien exécutée.

La réalisation de ce test utilise une classe de synchronisation `ThreadBarrier` de type *barrière*, présentée Prog. 18.1 p201. Elle comporte une méthode `synchronize()` qui permet d'attendre qu'un certain nombre de threads l'aient atteinte.

```
1  [...]
2  public class ThreadBarrier {
3      final Object lock = new Object();
4      int count, i;
5      boolean released;
6
7      public ThreadBarrier(final int count) { init(count);  }
8      private void init(final int count) {
9          this.count = count;
10         this.i = 0, this.released = false;
11     }
12     // Wait until 'count' threads have reached this barrier or
13     // the specified 'timeout' expires.
14     public boolean synchronize(final long timeout) throws InterruptedException {
15         synchronized(lock) {
16             if (++i >= count) {
17                 lock.notifyAll();
18                 released = true;
19             }else{
20                 lock.wait(timeout);
21             }
22         }
23         return released;
24     }
25 [...]
26 }
```

Prog. 18.1 – La classe `ThreadBarrier`.

L'utilisation de cette classe pour réaliser notre test pourrait naïvement s'écrire :

```
1  public void testAsynchronousInvocation() {
2      jaya.mandala.util.ThreadBarrier barrierProxy =
3                                      new jaya.mandala.util.ThreadBarrier(2);
4      FutureClient future = barrierProxy.rami_synchronize(TIMEOUT);
5      boolean ok1 = false, ok2 = false;
6      try{
```

```
7           ok1 = barrierProxy.synchronize(TIMEOUT);
8           Assert.assertTrue("Started", future.isStarted());
9           ok2 = ((Boolean) future.waitForResult(TIMEOUT)).booleanValue();
10      }catch(Throwable t) {
11          Assert.fail("An exception occurs: " + t.getMessage());
12      }
13
14      Assert.assertTrue("Asynchonous invocation", ok1 && ok2);
15  }
```

On cherche en effet à vérifier que deux threads s'exécutent en concurrence. L'appel asynchrone `rami_synchronize()` est donc censé invoquer la méthode `synchronize()` dans une thread dédiée. Dans le même temps, la thread courante réalise le même appel de manière synchrone. Si ce dernier appel retourne la valeur booléenne `true`, alors, la barrière a bien été traversée par deux threads, et le test passe. Dans le cas contraire, l'utilisation d'un faible *timeout* (5 secondes) évite de bloquer la suite des tests tout en laissant le temps à nos threads de traverser la barrière. Le choix de ce délai est empirique reflétant le problème rencontré dans les tests automatisés de programme concurrents : l'indéterminisme [128].

Rappelons que nous cherchons à tester notre implémentation des appels de méthodes semi-transparents, ce qui implicitement teste aussi l'asynchronisme de la méthode `call()` de la référence asynchrone associée.

Le problème de cette méthode de test est de ne pas tenir compte de la politique asynchrone (*cf.* §15.2.3.1 p167) utilisée : pour qu'une barrière puisse jouer son rôle, elle doit être traversée par plusieurs threads. Elle ne peut donc pas être associée à une politique asynchrone qui implémente une sémantique non-concurrente. Pire, comme l'appel `barrierProxy.synchronize()` délègue à la référence asynchrone le soin de faire l'appel, si la politique asynchrone implémente une sémantique non-concurrente, alors, la méthode `synchronize()` étant bloquante (`wait()`), le deuxième appel ne sera jamais pris en compte. La thread bloquée ne sera jamais réveillée (l'utilisation d'un délai d'expiration, permettra au test de terminer malgré tout).

Par conséquent, il faut modifier notre méthode de test pour qu'elle associe une politique "threadée" à notre barrière en utilisant une implémentation de l'interface `AsynchonousPolicy` telle que `ThreadedPolicy`. Celle-ci est retournée par le *factory* `ThreadedAPFactory`, une implémentation de l'interface `AsynchronousPolicyFactory` utilisée pour paramétrer la politique associée aux références asynchrones créées par la classe `ARFactory`. Nous écrirons donc :

```
1  public static final AsynchronousPolicyFactory apf = new ThreadedAPFactory();
2  [...]
3  public void testAsynchronousInvocation() {
4      // Must use a concurrent policy for locks !
```

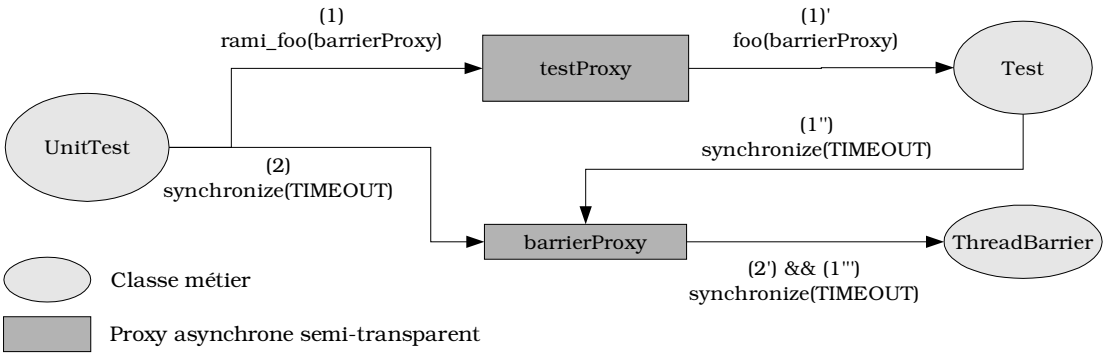


Fig. 18.1 – Illustration de l'architecture de classes utilisée pour réaliser le test unitaire d'un appel asynchrone utilisant RAMI.

```
5       Framework.Factory originalFactory = Framework.getFactory();
6       Framework.Factory threadedFactory = new ARFactory(apf);
7       Framework.Factory newFactory = new ARPFactory(originalFactory,
8                                                     threadedFactory);
9       jaya.mandala.util.ThreadBarrier barrierProxy =
10                              new jaya.mandala.util.ThreadBarrier(2, newFactory);
11
12      testAsynchronousInvocation(barrierProxy);
13  }
```

La méthode `testAsynchronousInvocation()` est surchargée avec un argument de type `jaya.mandala.util.ThreadBarrier`. C'est cette méthode qui contiendra le test proprement dit. Notons l'instanciation et la spécification d'un `ARPFactory` en ligne 7 et 9 lors de la construction du proxy asynchrone : le type de référence par défaut (et sa politique associée) pourra être gardé grâce à l'utilisation d'une paire de références (*cf.* §15.3 p172). Elle permettra d'avoir des barrières distantes lorsque le *factory* d'origine construit des références sur des *stored objects*.

Comme nous cherchons implicitement à tester toutes les implémentations de l'interface `AsynchronousReference` (y compris `StoredObjectReference` associée à un conteneur actif distant), ainsi que l'ensemble des politiques asynchrones, nous allons déléguer à une autre classe, nommée `TestClass`, le soin de faire l'appel à la méthode `synchronize()` comme illustré par la figure Fig. 18.1 p203. Ainsi, différentes politiques asynchrones pourront être testées. Cette classe contient la méthode `foo()` suivante :

```
1   public boolean foo(jaya.mandala.util.ThreadBarrier barrierProxy)
2           throws InterruptedException{
3       return barrierProxy.synchronize(TIMEOUT);
4   }
```

Notre test consiste donc à vérifier que l'appel à la méthode `rami_foo()` d'un proxy asynchrone semi-transparent d'une instance de la classe `TestClass` est bien asynchrone ce qui s'écrit :

```
1  import jaya.mandala.util.ThreadBarrier;
2  ...
3  void testAsynchronousInvocation(ThreadBarrier barrierProxy) {
4      FutureClient future = testProxy.rami_foo(barrierProxy);
5      boolean ok1 = false, ok2 = false;
6      try{
7          ok1 = barrierProxy.synchronize(TestClass.TIMEOUT);
8          Assert.assertTrue("Started", future.isStarted());
9          Boolean wrapper = (Boolean) future.waitForResult(TestClass.TIMEOUT)
10         ok2 = wrapper.booleanValue();
11     }catch(Throwable t) {
12         Assert.fail("An exception occurs: " + t.getMessage());
13     }
14
15     Assert.assertTrue("Asynchonous invocation of TestClass.foo()",
16                       ok1 && ok2);
17 }
```

Dans ce code, `testProxy` est un proxy asynchrone semi-transparent qui est initialisé par les classes filles.

Ce test unitaire est passé avec succès par toutes nos implémentations. Nous considérons notre test comme valide puisqu'il fait échouer une implémentation synchrone[3] de l'interface `AsynchonousPolicy`[4].

De nombreux tests unitaires ont été écrits : test de sérialisation, test d'annulation d'appel asynchrone, test d'égalité, test des collections ; le tout en version locale et distante. Pour de plus amples informations, le lecteur peut se référer au fichier `build.xml` utilisé par `Ant` et au répertoire `test` dans l'arborescence de Mandala [214].

18.4 Pertinence de nos modèles

18.4.1 Prise en charge de la concurrence

Le concept de référence asynchrone s'adapte particulièrement bien aux programmes orientés objets. En particulier, l'analogie traditionnelle établie entre les

[3]Cette implémentation permettra en §19.1.1 p211 d'évaluer le surcoût de la réflexion.

[4]Le lecteur trouvera le code correspondant dans la classe `mandala.rami.impl.Synchronous-Semantic`.

appels de méthodes et le passage de message reste vraie dans le cadre asynchrone. L'extension au cas distant n'engendre pas de difficulté conceptuelle particulière. C'est probablement ce qui explique la relative facilité du développement d'applications concurrentes (et/ou distribuées) qui utilisent ce paradigme (*cf.* §20 p227 pour des exemples de code).

Notons que d'autres paradigmes sont employés pour tirer partie de la concurrence. Citons par exemple le modèle *send()/receive()* employé dans le domaine du parallélisme (MPI [93]), et le modèle *publish/subscribe* utilisé par la bibliothèque *Java Messaging Service* [147] très proche du modèle événementiel présenté en §12.3.1 p126.

Il est aussi important de noter que dans la plupart des applications Java, aucun modèle particulier n'est utilisé. C'est ainsi que l'on retrouve souvent le squelette de code suivant :

```
new Thread() {
    public void run() {
        // business code
    }
}.start();
```

Celui-ci est peu extensible car il casse le paradigme classique d'appel de méthode : si des paramètres doivent être passés, une nouvelle classe qui hérite de `Thread` ou qui implémente `Runnable` doit être développée. Celle-ci devra être instanciée chaque fois qu'un nouveau paramètre devra être utilisé. Il nous semble plus naturel de définir la méthode métier avec un nom approprié plus explicite que `run()`.

Prenons pour exemple concret l'implémentation suivante d'un serveur :

```
ServerSocket ss = new ServerSocket();
while(true) {
    Socket socket = serverSocket.accept();
    new Thread() {
        public void run() {
            serve(socket);
        }
    }.start();
}
```

La politique asynchrone utilisée – ici, une thread par appel – est liée à la sémantique recherchée : libérer la thread principale pour pouvoir traiter d'autres clients.

L'utilisation d'un proxy asynchrone semi-transparent conduit à la version suivante :

```
ServerSocket ss = new ServerSocket();

```

```
3  // Creates an asynchronous proxy on a 'Service' instance
4  // which contains the serve() "business" method
5  jaya.Service service = new jaya.Service();
6  while(true) {
7      Socket socket = serverSocket.accept();
8      service.rami_serve(socket);  // asynchronous call
9  }
```

Ceci est beaucoup plus naturel et plus efficace : la politique asynchrone peut utiliser un *thread pool* sans modification syntaxique du code métier. La spécification de la politique est en effet réalisée au moment de l'instanciation du *proxy* :

```
1  ...
2  // Use a ThreadPooled asynchronous policy
3  AsynchonousPolicyFactory asyncPolicyFactory = new ThreadPooledAPFactory();
4  // Creates an asynchronous proxy on a 'Service' instance
5  // which contains the serve() "business" method
6  jaya.Service service = new jaya.Service(new ARFactory(asyncPolicyFactory));
7  ...
```

Il est ainsi très simple de changer la politique, ce qui permet de réaliser des mesures pour améliorer les performances de l'application sans altération du code. L'absence de thread dans le code nous semble être une bonne chose : le développeur utilise seulement le paradigme appel de méthode (synchrone ou asynchrone) qu'il connaît particulièrement bien.

18.4.2 Distribution

Le concept des conteneurs actifs nous semble être très prometteur : il identifie clairement la "localisation" d'un objet. Si dans le cas local, la machine virtuelle est le conteneur standard, dans le cas distant, on utilise des entités plus ou moins masquées (`Node` de ProActive[41], `Context` des aglets [53], `RemoteRef` de RMI [191], ...). L'abstraction *conteneur distant* remet l'aspect distant à sa place : si une machine virtuelle ne répond plus, le client peut gérer l'erreur de manière globale. Dans RMI, l'aspect distant est lié à l'objet, non à son conteneur ; le client doit donc gérer les erreurs indépendamment, pour chaque objet RMI. Cela entraîne un grand nombre de contraintes. Il est facile de s'en rendre compte sur un exemple très simple : le programme `HelloWorld`. La figure FIG. 18.2 p207 présente une comparaison du code nécessaire à l'écriture de la partie serveur d'un tel programme. La figure FIG. 18.3 p208 compare le code d'un client utilisant RMI ou Mandala.

L'utilisation de RMI pour l'implémentation d'un objet distant requiert donc plusieurs étapes :

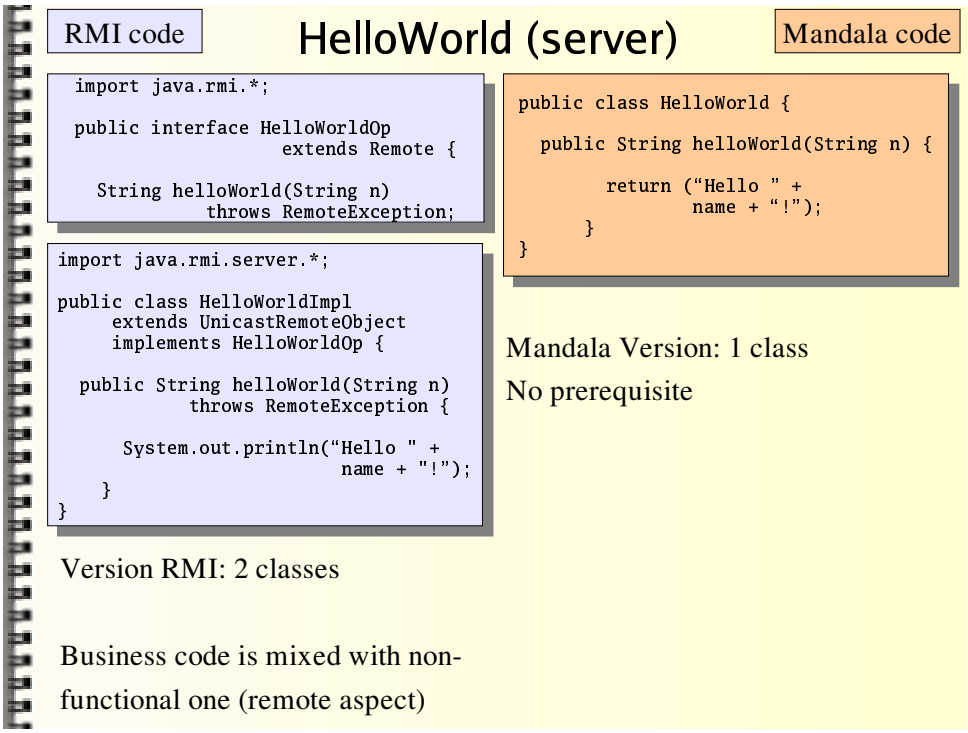


Fig. 18.2 – Comparaison du code nécessaire à l'écriture d'un serveur type `HelloWorld` en utilisant RMI et Mandala.

- définir l'interface distante ;
- implémenter cette interface ;
- compiler le tout ;
- générer la classe dont les instances serviront de *stub*[5] ;

En outre, il ne suffit pas d'implémenter l'interface `java.rmi.Remote` pour rendre une classe distante. Un code doit s'occuper de l'aspect distant : c'est le fragment serveur(*cf.* §9.1 p79). Généralement, une classe RMI hérite de `java.rmi.server.UnicastRemoteObject` qui fournit un tel code. Cependant, si par oubli, elle ne le fait pas, le code reste valide, mais le comportement n'est pas celui attendu[6].

La mise à disposition dans un service de nommage est commune aux deux technologies. On préfèrera JNDI [144] pour le support d'environnements hétérogènes (LDAP [220], rmiregistry [191], COS [160], etc.).

Pour le client, la récupération d'un proxy fait appel à un service de nommage. Cette étape est commune aux deux technologies. Notons toutefois que dans Mandala, le proxy est clairement identifié comme tel : le préfixe `jaya` impose au développeur de

[5]Cette phase ne sera plus nécessaire avec le JDK v1.5 grâce à l'utilisation des proxys dynamiques.

[6]Erreur rencontrée lors de travaux pratiques sur le développement d'agents mobiles en Java/RMI : les élèves ont eu beaucoup de mal à trouver l'origine du mauvais comportement de leur agents !

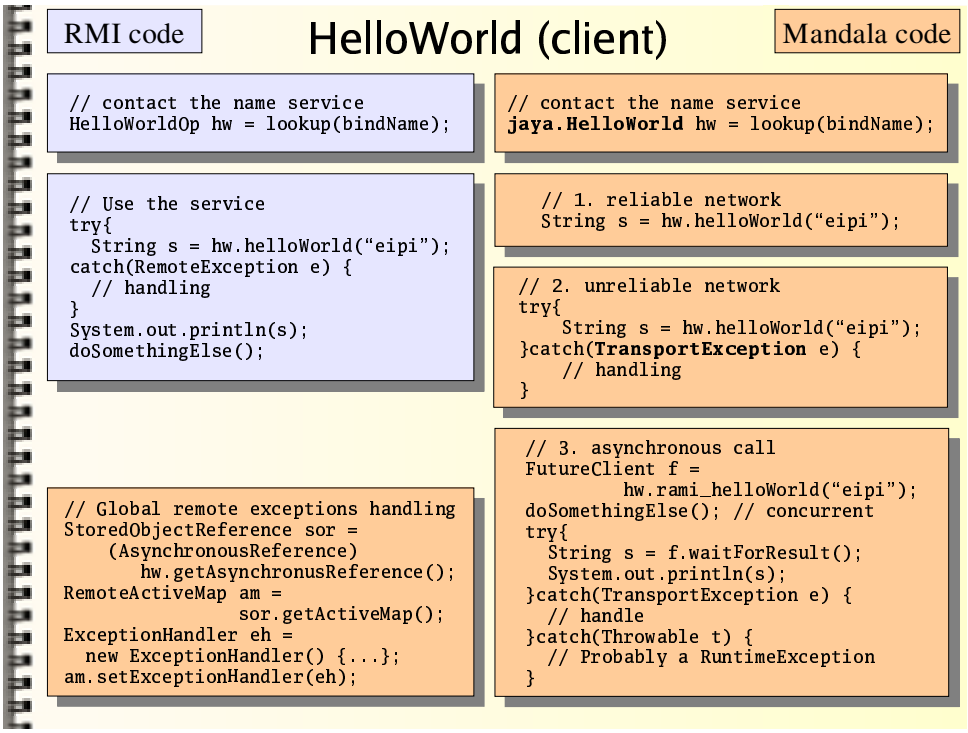


FIG. 18.3 – Comparaison du code client d'un serveur type `HelloWorld` en utilisant RMI et Mandala.

prendre conscience de l'aspect distant et potentiellement asynchrone de ses appels.

L'utilisation d'un objet RMI est contrainte par la levée éventuelle de l'exception `java.rmi.RemoteException`. Or, au niveau syntaxique, un appel distant n'est pas différent d'un appel standard. C'est pourquoi dans Mandala, les exceptions réseaux sont non-contrôlées de type `mandala.jacob.remote.TransportException`. Elles contiennent toutes les informations nécessaires à leur gestion (objet cible de l'appel, méthode invoquée, arguments, etc.). De plus, les possibilités offertes par Mandala sont nombreuses : utilisation d'un callback, gestionnaire d'exception global, migration de code, et surtout appel asynchrone.

Concernant ce dernier aspect, dans la plupart des plate-formes qui fournissent le paradigme *appel de méthode asynchrone distant* (*cf.* §9 p79), l'asynchronisme est de type *côté serveur* (*cf.* §12.1 p123). S'il reste indispensable pour garantir à l'appelant que son appel distant a bien été pris en compte, ce type d'asynchronisme peut être à l'origine de problèmes de performance. Dans Mandala, non seulement la politique asynchrone utilisée par le conteneur est paramétrable, mais encore, les appels de méthodes asynchrones distants peuvent être de tout type. Le tableau TAB. 18.1 p209 présente les différents types d'asynchronisme possible[7] pour l'accès distant à

[7]Un asynchronisme *côté client* seulement reviendrait à rendre la méthode `ActiveMap.call()`

Type	Politique	Référence	Code
côté serveur	conteneur	$RSOR(o)$	`return new SORFactory(activeMap);`

complet	client : X
	serveur : conteneur
		$< AR_X, RSOR > (o)$	`xF = new ARFactory(XAPFactory);`
			`rsorF = new SORFactory(activeMap);`
			`return new ARPFactory(xF, rsorF);`

complet	client : X
	serveur : Y
		$< AR_X, RSOR, AR_Y > (o)$	`xF = new ARFactory(XAPFactory);`
			`rsorF = new SORFactory(activeMap);`
			`yF = new ARFactory(YAPFactory);`
			`rsorFyF = new ARPFactory(rsorF, yF);`
			`return new ARPFactory(xF, rsorFyF);`

Tab. 18.1 – Type d'asynchronisme dans Mandala.

un objet o. Pour chacun d'eux, le type de référence et le code permettant l'obtention du *factory* utilisé par la classe `mandala.rami.Framework` (*cf.* §18.2 p200) pour sa création sont présentés.

18.5 Bilan

Mandala permet aux développeurs de s'affranchir des contraintes imposées par RMI : l'aspect *dynamique* apporte un développement centré sur l'aspect métier des classes développées. Par ailleurs, Mandala offre des possibilités que ne permettent pas directement Java et RMI ; nous l'avons vu sur de nombreux exemples. La *flexibilité* du système permet très facilement d'adapter le comportement de la plate-forme aux objectifs recherchés dans l'application.

synchrone ce qui ne présente aucun intérêt : si le client utilise un asynchronisme côté client, il peut aussi utiliser un asynchronisme *complet*.

Chapitre 19

Mesures de performances

Dans ce chapitre les performances des implémentations de nos deux concepts, *les références asynchrones* et *les conteneurs actifs* vont être étudiées. Dans un premier temps, les analyses de RAMI vont mettre en évidence, dans le cas local, l'influence des différentes politiques sur la latence perçue par le client d'un appel asynchrone et sur les performances globales d'une application. Ensuite, l'aspect distant sera pris en compte en étudiant JACOb, et les effets du protocole utilisé sur la latence d'un appel de méthode asynchrone distant, ainsi que sur l'exécution totale de l'application seront présentés. Finalement, le bénéfice de l'utilisation des conteneurs actifs distant pour le partage des ressources sera considéré.

19.1 Performances de RAMI

Analyser les performances d'un appel de méthode asynchrone n'est pas commode : un tel appel ne peut être comparé à sa version synchrone. Néanmoins, nous avons effectué un certain nombre de mesures pour tenter d'évaluer le surcoût induit par l'utilisation de la réflexion (qui assure le dynamisme, *cf.* §9.1 p79) et le comportement de la plate-forme sur un exemple concret.

19.1.1 Surcoût de la réflexion

Nous allons étudier le surcoût induit par l'utilisation de la réflexion dans les invocations de méthodes asynchrones sur des appels (locaux) vides (*i.e.* avec un corps de méthode réduit à sa plus simple expression). La table 19.1 présente le temps moyen nécessaire pour invoquer une méthode vide en utilisant différentes politiques asynchrones avec différents JDKs. Les mesures pour l'implémentation synchrone (mentionnée en §18.3 p204) sont présentées à titre indicatif. Les tests ont été effectués 100 000 fois sur un système GNU/Linux 2.4/Bi-Pentium III 450 MHz avec

JDK	Synchrone	Concurrente		Non-concurrente	
		Threaded	ThreadPooled	Fifo	Random
1.2	137.22	**60.263**	26.837	23.953	23.861
1.3 -c	89.12	911.833	48.130	**20.684**	**20.746**
1.3 -s	95.65	905.551	50.837	21.221	21.398
1.4 -c	87.43	72.499	24.950	22.260	22.791
1.4 -s	**77.62**	74.788	**24.299**	22.210	22.748
`-c = java -client; -s = java -server`					
Les meilleurs temps sont en **gras**.					

TAB. 19.1 – Coût de la plate-forme RAMI en nanosecondes par appel.

les différentes machines virtuelles de Sun[1]. Le JDK v1.2 utilise une JVM classique tandis que les JDK v1.3 et JDK v1.4 utilisent la technologie HotSpot [193]. Le test invoque la méthode vide un millier de fois avant que les mesures ne soient réalisées pour prévenir tout effet de cache ou de compilation à la volée[2].

Notons que l'utilisation d'une sémantique synchrone[3] induit un surcoût considérable par rapport aux appels standards. Les mesures pour ces derniers ne sont pas présentées dans la table puisque le temps moyen de ces appels est proche de zéro (le minimum est de 3 ms pour 100 000 appels pour le JDK v1.2 et le maximum, de 38 ms pour le JDK v1.4 avec l'option `-client`). Précisons que les méthodes vides peuvent être optimisées de manière agressive (*inline* en `NOP`, *No Operation*) par les compilateurs *just in time*, tandis que le travail à réaliser pour qu'une invocation de méthode RAMI soit dynamique et asynchrone n'est pas négligeable : un objet `FutureClient` est créé pour chaque invocation de la méthode `AsynchronousReferenceImpl.call()` en utilisant un *factory*. De plus, la création de nombreux objets à faible durée de vie comme les *futures* accroît le travail du ramasse-miettes diminuant ainsi les performances globales. Ce dernier point est très facilement mis en évidence en augmentant la taille du tas alloué à la machine virtuelle à l'aide des options `-Xms` et `-Xmx` de l'exécutable `java` : le même test pour une sémantique "synchrone" avec la ligne de commande (JDK v1.4) :

```
java -client -Xms256m -Xmx512mx
```

apporte une amélioration de 35% avec une moyenne de 56.72 ns/appel. Nous expérimenterons deux techniques différentes pour éviter la création d'objets à faible

[1]Les sources des classes correspondantes sont dans le répertoire `Examples/src/emptyBench/` de l'arborescence de Mandala [214].

[2]On suppose que c'est suffisant.

[3]Pour rappel, elle n'est pas conforme à la spécification et ne passe pas les tests unitaires (*cf.* §18.3 p204).

durée de vie : la réutilisation explicite et la réutilisation implicite d'instance. La première technique fait appel à un *pool* d'objets *futures* et demande la participation du développeur. La seconde utilise les références molles (*soft* ou *weak reference*) du *package* `java.lang.ref` pour automatiser la réutilisation. Ces deux techniques seront développées dans des implémentations de l'interface `FutureFactory` différentes ne conduisant pas à des incompatibilités de versions.

Les colonnes "ThreadPooled" et "Threaded" montrent le surcoût des sémantiques concurrentes. Comme prévu, la création d'une thread par appel induit une lourde charge. Par ailleurs, nous avons observé[4] avec ce test le comportement du système de gestion des threads des différentes JVM : le test avec le JDK v1.4 passe 30% du temps dans l'espace noyau, et 46% dans l'espace utilisateur avec une moyenne de 1500 threads créées par seconde. Nous pensons que le temps passé en espace noyau est lié au changement de contexte des threads, tandis que celui en espace utilisateur est lié à l'exécution du code RAMI et du ramasse-miettes. Lors du test avec le JDK v1.3, seulement 3% du temps CPU était en espace noyau, 5% en espace utilisateur, tandis que le nombre moyen de threads créées par seconde ne dépassait pas les 200. La machine virtuelle HotSpot du JDK v1.3 paraît très immature tant la gestion des threads, des verrous et de la mémoire semble inefficace.

Les colonnes "Fifo" et "Random" présentent le surcoût des sémantiques non-concurrentes : les appels sont placés dans une file, ce qui requiert un minimum de synchronisation.

Globalement, la machine virtuelle HotSpot du JDK v1.3 est particulièrement inefficace.

Utiliser RAMI de la manière décrite ci-dessus n'est pas l'approche normale de toute façon. D'une part, les méthodes vides ne sont pas courantes dans les applications réelles, mais surtout, elles n'ont généralement pas à être invoquées de manière asynchrone. Nous allons donc effectuer des mesures de performances sur un exemple réel.

19.1.2 Analyse de performance de RAMI sur un exemple réel : calcul du nombre π

Nous avons testé RAMI pour le calcul des décimales du nombre π. L'algorithme utilisé est celui trouvé par Bailey, Borwein et Plouffe [24]. La formule utilisée est :

$$\pi = \sum_{i=0}^{\infty} f(i), \text{ avec } f(i) = \frac{1}{16^i}\left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6}\right)$$

[4]Ces observations n'apparaissent pas dans le tableau précédent.

Cette formule est implémentée dans une classe nommée `PiComputer`[5] qui contient, entre autres, la méthode :

```
1 BigDecimal compute(int start, int end);
```

Ainsi, pour calculer les n premières décimales de π, il suffit de créer une instance[6] de cette classe et d'invoquer la méthode `computer.compute(0, n)` qui calcule la somme $S_n = \sum_{i=0}^{n} f(i)$.

L'utilisation de RAMI et de son asynchronisme permet d'exploiter le parallélisme sous-jacent : le calcul original peut être découpé en p sous-sommes $s_1, \ldots, s_p$ où p dépend du nombre de processeurs utilisables et

$$s_i = \sum_{k=b_i}^{e_i} f(k) \text{ tel que } S_n = \sum_{i=1}^{p} s_i$$

avec

$$b_i = \begin{cases} 0 & \text{si } i = 1, \\ i.\lfloor \frac{n}{p} \rfloor & \text{sinon.} \end{cases} \text{ et } e_i = \begin{cases} n & \text{si } i = p, \\ b_i + \lfloor \frac{n}{p} \rfloor - 1 & \text{sinon.} \end{cases}$$

La méthode `main()` invoque p fois la méthode `compute()` de manière asynchrone en spécifiant les paramètres b_i et e_i calculés par une méthode nommée `distribute()` que nous ne présenterons pas. Si plusieurs processeurs sont disponibles, les sous-sommes s_i peuvent être calculées en parallèle.

Avant chaque test, des appels factices sont effectués pour éviter les effets liés au cache ou au compilateur *just in time*. La méthode `System.gc()` est aussi systématiquement invoquée avant un test pour permettre au ramasse-miettes de nettoyer la mémoire. Enfin, la taille du tas de la JVM (JDK v1.3 d'IBM) est spécifiée avec un minimum de 128 Mo (option `-Xms128m`) et un maximum de 256 Mo (option `-Xmx256m`). Etant donné le grand nombre de paramètres qui entrent en ligne de compte dans les performances d'une application Java, les résultats présentés ne donnent qu'un aperçu général des performances qu'il est possible d'obtenir avec RAMI.

Le test a été effectué sur une machine parallèle quadri-processeurs AIX IBM R6000 SP3 375 Mhz pour le calcul de 1000 décimales. Sur le système AIX, la variable d'environnement `AIXTHREAD_MNRATIO` spécifie le nombre de threads utilisateur par thread noyau. Comme toutes les threads impliquées dans cet exemple sont des consommatrices de ressources CPU, un ratio de `1:1` a été utilisé.

[5]Le code de cette étude est disponible dans le répertoire `Examples/src/pi/` de l'arborescence de Mandala [214].

[6]Le lecteur pourrait se demander pourquoi la méthode `compute` n'est pas déclarée `static`. En fait, le constructeur de `PiComputer` permet de préciser l'échelle (*scale*) nécessaire à la classe `java.math.BigDecimal` qui est utilisée.

Subsums	Direct	Synchronous	Concurrency		SingleThreaded	
			Threaded	ThreadPooled	Fifo	Random
1	198201	**198066**	197893	197890	**197862**	**197881**
2	198170	198107	101148	101268	197922	197944
3	198174	198122	68537	68571	197962	197963
7	198195	198316	**51240**	52702	198117	198153
11	198194	198514	52231	56169	198314	198334
31	198168	199394	51326	**51791**	199235	199250
51	**198166**	200284	55237	55974	200103	200121
Les meilleurs temps sont en **gras**						

Tab. 19.2 – Temps en millisecondes nécessaire au calcul des 1000 premières décimales de π sur un noeud quadri-processeur en utilisant RAMI.

L'invocation directe de `PiComputer.compute(0,1000)` dure en moyenne 3:18.20 (3 minutes et 18.20 secondes, moyenne sur 7 exécutions). La table 19.2 présente le temps nécessaire au calcul des 1000 premières décimales de π en fonction du nombre p de sous-sommes[7]. La figure 19.1 est une représentation graphique de ces résultats.

Nous pouvons faire plusieurs remarques :

- le meilleur temps est mesuré pour $p \geq 4$ comme prévu (machine quadri-processeurs) produisant un *speedup*[8] S de :

$$S = \frac{BestSequentialTime}{BestParallelTime} = \frac{198166\ (\texttt{Direct},\ \texttt{p} = 51)}{51240\ (\texttt{Threaded},\ \texttt{p} = 7)} = \frac{3:18.17}{0:51.24} \approx 3.87$$

  et une efficacité E de :

$$E = \frac{Speedup}{ProcessorsNumber} = \frac{3.87}{4} = 0.9675$$

- le temps pour n'importe quelle sémantique (y compris la sémantique synchrone) est parfois meilleur qu'une invocation directe ;
- la politique asynchrone `Threaded` offre les meilleures performances[9].

La première remarque révéle qu'une sémantique concurrente est plus efficace comme prévu. La seconde remarque semble indiquer que RAMI n'introduit pas un surcoût très important malgré les opérations supplémentaires induites par son utilisation comme nous l'avons vu dans la section §19.1.1 p211. Nous avions suspecté le

[7]La classe `TestPerfPiComputer` est utilisée à cet effet.

[8]Selon la définition du *speedup* où le meilleur temps séquentiel est celui de l'invocation directe de la méthode `compute()` sans la couche RAMI.

[9]La différence avec `ThreadPooled` peut sembler minime, mais elle est suffisamment rencontrée pour qu'elle devienne significative.

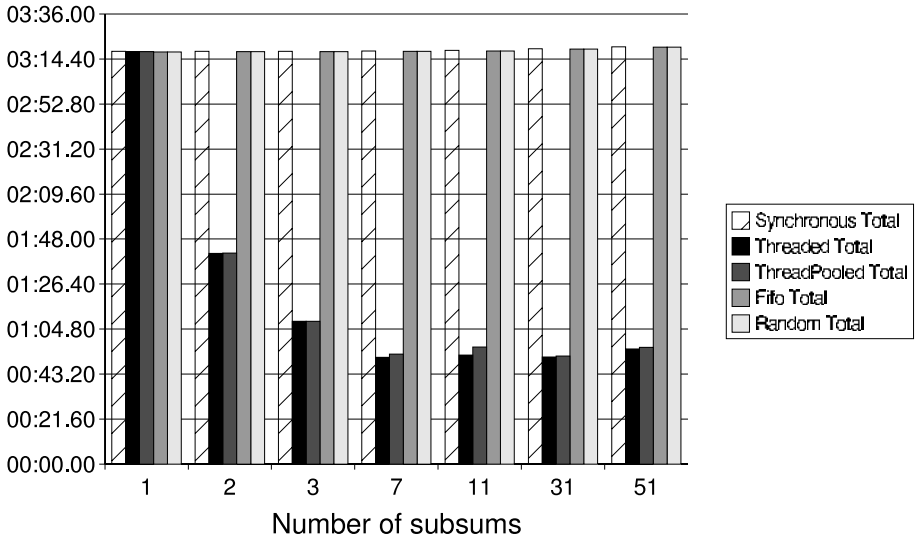


FIG. 19.1 – Temps nécessaire au calcul des 1000 premières décimales de π sur un noeud quadri-processeur d'une machine parallèle en utilisant RAMI.

JIT d'être à l'origine de cette singularité par l'introduction de bruit dans nos *benchmarks*. Cependant, après une exécution en mode interprété (option `-Xint`), nous obtenons le même comportement étrange. On peut supposer que le système d'exploitation (allocations mémoires) ou même le matériel (cache processeurs) en sont responsables, mais de plus amples investigations semblent nécessaires pour l'expliquer clairement.

Notons que dans le cas où $p = 1$, il n'y a qu'une seule invocation asynchrone et le surcoût engendré par RAMI est minimum. Il est donc normal d'observer un temps comparable à l'invocation directe. Le surcoût devient perceptible lorsque p croît, particulièrement dans la version synchrone.

La dernière remarque semble montrer que la politique `Threaded` est la plus efficace en terme de temps total de calcul (le temps entre le lancement de l'application, et l'arrêt de l'application). Toutefois, si l'on mesure la latence de l'appel à la méthode `call()`, *i.e.* le temps nécessaire avant de recevoir un *future*, la politique *Threaded* est la moins efficace. La table 19.3, et sa représentation graphique associée 19.2 nous le montre clairement (la latence de la sémantique synchrone est égale au temps de calcul et n'est donc pas représentée graphiquement). Créer une thread à chaque invocation de méthode asynchrone a un coût. Comme on pouvait s'y attendre, la politique `ThreadPooled` est donc le meilleur compromis.

Subsums	Synchronous	Concurrency		SingleThreaded	
		Threaded	ThreadPooled	Fifo	Random
1	197978	18	0	0	0
2	99009	9	2	0	6
3	66011	6	1	0	3
7	28318	2	0	0	1
11	18038	11	0	0	1
31	6429	122	0	0	0
51	3925	91	0	0	0

Tab. 19.3 – Latence en millisecondes de l'appel de méthode.

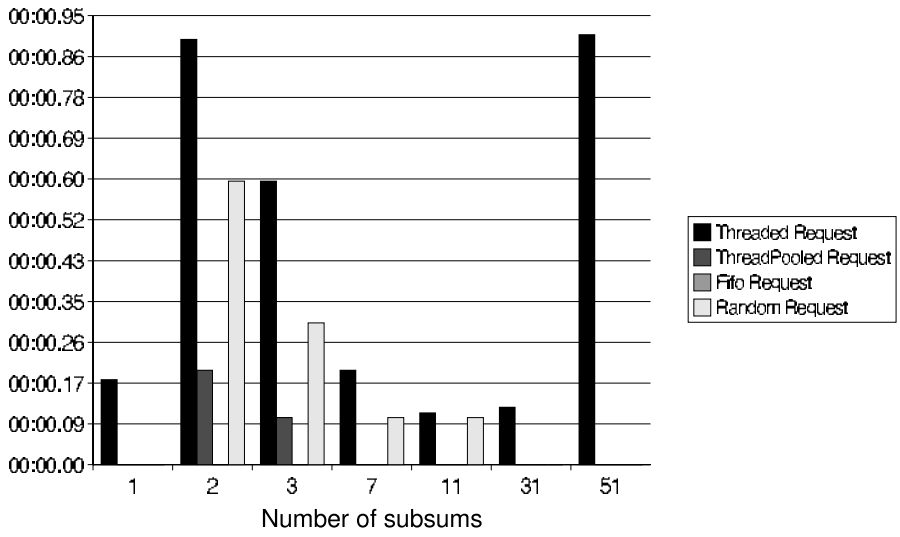


Fig. 19.2 – Latence d'un appel de méthode asynchrone (une latence de zéro possède une barre de longueur nulle et n'est donc pas représentée).

19.2 Performance de JACOb

Même si JACOb propose des implémentations distantes du paquetage *collection* de l'API Java, nous allons réduire notre analyse à la vocation première de nos conteneurs actifs : rendre n'importe quel objet accessible à distance dynamiquement. Par conséquent, nous allons utiliser un protocole comparable à celui de la section précédente : le surcoût lié à JACOb sera comparé dans un premier temps à celui de RMI sur des appels de méthodes vides. Une deuxième analyse mettra en œuvre le calcul des décimales de π.

19.2.1 Mesures sur des appels vides

Le code utilisé pour l'étude du surcoût de RAMI présentée en §19.1.1 p211 permet de spécifier une référence distante sur un *stored object*. C'est donc celui que nous allons exécuter. Cependant, dans le cadre distant, nous allons faire varier les protocoles de communication utilisés au lieu des politiques asynchrones. Le test a été réalisé entre deux systèmes GNU/Linux 2.4/Pentium III 450 mhz, un mono-processeur et un bi-processeurs, reliés par un réseau Ethernet 100 Mb/s *non dédié*, en période de relative inactivité[10]. Le conteneur actif est exécuté sur le système bi-processeurs. Pour information, la commande 'ping -nU -c 100' du client vers le serveur affiche l'information suivante :

```
rtt min/avg/max/mdev = 0.160/0.240/0.281/0.028 ms
```

Par ailleurs, le transfert d'un fichier de 125 méga-octets[11] par FTP entre les deux répertoires temporaires locaux (/tmp) met en moyenne sur 14 transferts, 25,9 secondes, ce qui correspond à un débit de 38,61 Mbits/s. La table 19.4 montre les temps nécessaires pour réaliser, avec le JDK v1.4 de Sun, 10 000 appels de méthodes vides sur un *stored object* distant. La figure 19.3 est une représentation graphique de ces données. Différents protocoles sont testés :

- nos trois implémentations basées sur le *Generic Protocol Framework* (*cf.* §10.3.2 p107), à savoir RMI, UDP et TCP ;
- notre implémentation naïve RMI (*cf.* §10.3.1 p105) ;
- une version totalement réécrite qui utilise RMI directement.

Cette dernière version requiert une réécriture puisque la classe contenant la méthode vide (`emptyBench.EmtpyBench`) n'est pas conforme aux spécifications RMI

[10] Mois d'août en France.

[11] Depuis décembre 1998, le système des unités internationales a défini les unités couramment utilisées dans les systèmes informatiques et électroniques [12] : 1024 octets = 1 Kibi-octets. Par conséquent, aujourd'hui, une carte Ethernet 10 Mbits/s a donc un débit maximal de 10 000 000 bits/s et non de 1 048 576 bits/s.

	GPF			RMI	RMI Direct
	RMI	UDP	TCP		
Temps total	03:41.94	04:19.24	14:53.01	04:46.33	00:05.39
Temps par appel (ms)	22.19	25.92	89.3	28.63	0.54

Tab. 19.4 – Coût de 10 000 appels de méthode vide à distance dans la plateforme JACOb.

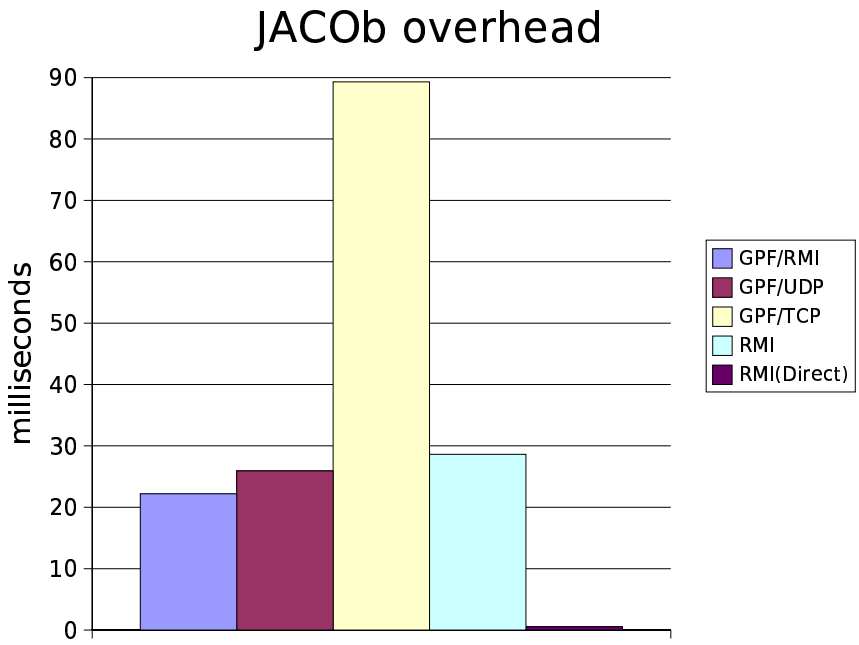


Fig. 19.3 – Coût de 10 000 appels de méthode à distance dans la plateforme JACOb.

(elle n'implémente pas `java.rmi.Remote` en particulier).

On observe que la version RMI directe obtient des performances sans commune mesures avec les autres versions. Là encore, la machine virtuelle, le compilateur ou le *runtime* RMI, sont capables de faire des optimisations agressives. Par exemple, la classe *squelette* (*cf.* §9.1 p79) générée par le compilateur `rmic` invoque directement la méthode `emtpy()` sans utiliser le mécanisme réflexif. Il est donc possible pour le JIT de rendre *inline* cette méthode, voire d'en éliminer le code puisqu'elle est vide.

Ces mesures mettent en lumière l'inefficacité de notre version TCP. Elle est liée à une implémentation à base de processus : chaque nouvelle connexion entraîne la création d'une thread. On retrouve ici les problèmes liés à la gestion de la concurrence (*cf.* §12.3 p126) inhérents aux objets distribués (*cf.* §10.2.4 p104) :

le modèle à base de *threads* est ici excessivement peu performant. A moins d'utiliser une bibliothèque de *threads* plus efficace, un modèle évènementiel semble[12] plus adapté. L'API `java.nio` [226] a été proposée à cet effet. Elle est disponible depuis le JDK v1.4 et permet de multiplexer les entrées/sorties. Comme nous l'avons mentionné en §10.3.2 p107. L'implémentation TCP a été conçue rapidement, sans chercher à obtenir de bonnes performances : c'est un prototype.

Soulignons que la version naïve utilisant RMI est moins efficace que l'implémentation RMI utilisant le GPF. Cela confirme la relative adéquation de l'architecture du GPF avec les implémentations distantes du paquetage *collection*. Notre implémentation UDP est 1,16 fois plus lente que la version RMI/GPF : elle est donc loin d'être optimale. Elle utilise en effet un modèle à base de processus – nettement plus performant que la version TCP – mais qui mériterait probablement d'être changé par un modèle événementiel utilisant le nouveau *package* `java.nio`.

Au vu de ces résultats, on peut se demander si le surcoût important observé (facteur 40 entre la version RMI/GPF et la version RMI directe) n'est pas le revers de la médaille lié au *dynamisme* qu'offre JACOb. Pour répondre à cette interrogation, nous devons prendre en considération un certain nombre d'éléments :

- les méthodes vides sont très rarement rencontrées dans les classes d'une application orientée objets ;
- les méthodes dont le corps est d'une complexité faible ont généralement peu d'intérêt à être invoquées de manière distante, encore moins de manière asynchrone.

Des mesures sur un exemple plus concret ont donc été réalisées.

19.2.2 Analyse de performance de JACOb sur un exemple réel : calcul du nombre π

En reprenant le calcul des décimales du nombre π comme exemple (*cf.* §19.1.2 p213), nous allons observer le comportement des différentes implémentations distantes du concept de conteneur actif. Le test[13] fait intervenir les mêmes machines que précédemment : le conteneur actif est lancé sur le bi-pentium III, le client est lancé sur la machine mono-processeur. JNDI est utilisé pour référencer le conteneur. Après insertion d'une instance de la classe `PiComputer`[14] dans le conteneur distant, le test commence par réaliser un certain nombre d'appels factices pour lisser les effets

[12] L'emploi du conditionnel est nécessaire en raison de la relation de dualité entre les systèmes à base de processus et les systèmes évènementiels que nous avons présentée en §12.3 p126.

[13] La classe `TestPerfRemotePiComputer` est utilisée.

[14] L'insertion est automatique lors de l'instanciation d'un proxy semi-transparent (*cf.* §16.2.3 p189) associé à une instance de `SORFactory` (*cf.* §2.2.1 p10).

Subsums	GPF			RMI
	RMI	UDP	TCP	
1	90	34	58	85
2	86	52	77	100
3	65	51	58	97
7	64	52	**56**	74
11	62	**28**	57	69
31	54	30	58	63
51	**47**	31	**56**	**56**
Les meilleurs temps sont en **gras**				

Tab. 19.5 – Latence moyenne en millisecondes d'un appel asynchrone distant *côté serveur* sous JACOb.

du JIT. Le test réel démarre juste après l'invocation de la méthode `System.gc()` pour éviter autant que possible l'intervention du ramasse-miettes pendant le test. Le tableau 19.5 présente la latence moyenne induite par un appel de méthode distant asynchrone *côté serveur* (*cf.* §12.1 p123) en fonction du nombre d'appels (le paramètre *subsums*) et du protocole utilisé[15]. La politique asynchrone *côté serveur* est celle du conteneur qui est par défaut une politique `ThreadPooled`.

Une représentation graphique de ces données, présentée sur la figure 19.4, montre clairement que notre architecture nommée *Generic Protocol Framework* est plus efficace que notre implémentation RMI naïve. Cette dernière (*cf.* §10.3.1 p105) est pourtant une implémentation qui s'efforce de suivre les recommandations de la documentation. Cela confirme donc l'intérêt de l'aspect *dynamique* de JACOb. En séparant l'aspect distant de la logique métier des objets, il est possible de gagner sur les deux tableaux : la logique métier peut être optimisée indépendamment de l'aspect distant et vice-versa.

La version UDP offre une alternative intéressante, puisque la latence est en moyenne divisée par un facteur proche de 2 par rapport à la version GPF/RMI.

Ces mesures imposent une remarque : même s'il est asynchrone côté serveur (*cf.* §12.1 p123), un appel de méthode distant a un coût qui est loin d'être négligeable, en particulier en Java. Ce problème est à l'origine de nos travaux sur la gestion de la concurrence (*cf.* §12 p123). Notons en outre que la plupart des plate-formes, qui revendiquent l'appel de méthode asynchrone distant comme paradigme de programmation des applications distribuées, ne mentionnent pas le type d'asynchronisme utilisé. Pour l'essentiel des travaux que nous avons étudiés, l'asynchronisme est de type serveur. Ils posent donc le problème de latence mis en évidence dans ce tableau.

[15]Le temps de calcul total n'a aucun intérêt ici : les résultats étant très similaires pour l'ensemble des protocoles, aucune information ne peut en être tirée.

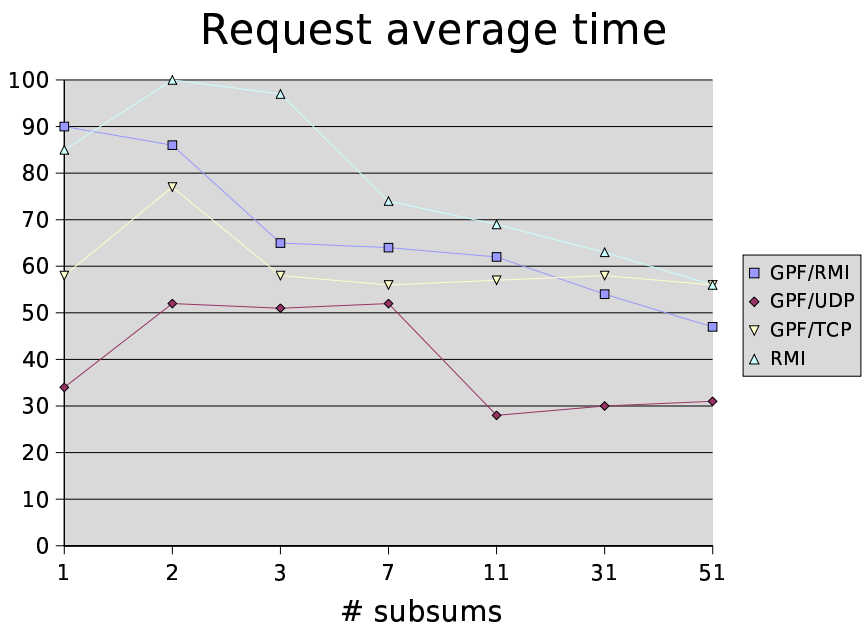


FIG. 19.4 – Latence d'un appel de méthode à distance asynchrone.

Nous proposons un asynchronisme dit *complet* fourni par le chaînage de références asynchrones présenté en §15.3 p172 afin de résoudre ce problème. En utilisant une paire de références de type $< \texttt{AR}_{\texttt{ThreadPooled}}, \texttt{SOR} >$ (`PiComputer`) on obtient un asynchronisme :

- côté client, assuré par une référence asynchrone locale associée à une sémantique concurrente ;
- côté serveur, assuré par une référence sur un *stored object*, dont la politique est celle du conteneur (`ThreadPooled` par défaut).

Les performances obtenues dans une telle configuration sont présentées dans la figure 19.5. Les courbes représentent la différence entre la latence d'un appel asynchrone côté serveur seulement et la latence d'un appel asynchrone complet. Un gain est obtenu lorsque les courbes sont *au-dessus* de l'axe des abscisses. Clairement, ce n'est pas le cas : nous observons même une augmentation de la latence la plupart du temps !

Une explication peut être trouvée dans notre implémentation de nos propres spécifications. Pour des raisons de modularité, nous avons usé (et probablement abusé) de paradigmes de conceptions particulièrement lourds comme les *factorys*, les *singletons* et autres *design patterns* [82]. Par exemple, notre implémentation des paires de références (la classe `AsynchronousReferencePairImpl` du *package* `mandala.rami.impl`) semble être particulièrement inefficace. Une étude plus approfondie mérite d'être réalisée afin de déceler les fragments du code qui sont les plus coûteux. Nous pensons qu'une telle étude, avec des outils adéquats (*profiler*) permettra de

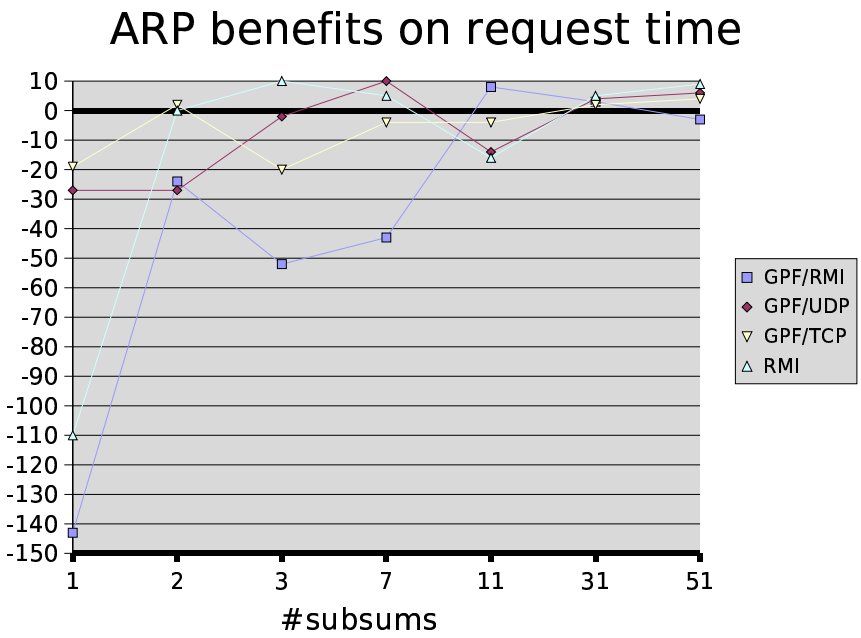


Fig. 19.5 – Gain obtenu par l'utilisation d'un chaînage de références asynchrones.

réduire sensiblement la complexité des appels asychrones complets, justifiant ainsi leurs mises en œuvre.

19.2.3 Partage de ressources

Nous avons vu en 10.5.2 que l'utilisation d'un conteneur actif permet de partager les ressources associées aux objets exportés puisque c'est le conteneur qui est la *partie serveur* (*cf.* 9.1) de chaque objet distant. Nous allons mesurer l'impact de ce partage sur les performances d'une application qui exporte de nombreux objets. Le test consiste à instancier p objets distants et à mesurer la quantité de mémoire consommée (en utilisant la méthode `Runtime.freeMemory()`) ainsi que le temps nécessaire à cette création. Les objets distants sont des instances de la classe `EmptyBench` pour les *stored objects* et de `EmptyBenchImpl` pour la version RMI vue précédemment[16]. Cela permet de minimiser au maximum l'effet de la partie métier (constructeur vide, pas de champ, une méthode vide). Les mesures ont été réalisées sur un système GNU/Linux bi-pentium III 450 Mhz avec le JDK v1.4 de Sun microsystem. La figure 19.6 présente la consommation mémoire en fonction du nombre d'objets distants créés.

Clairement, cette figure montre que rendre un objet distant est une opération coûteuse en mémoire : pour une seule instance de notre classe métier vide, environ

[16]Le code correspondant à ce test se trouve dans le fichier `SharingBench.java` du répertoire `Examples/src/emptyBench/` de l'arborescence de Mandala.

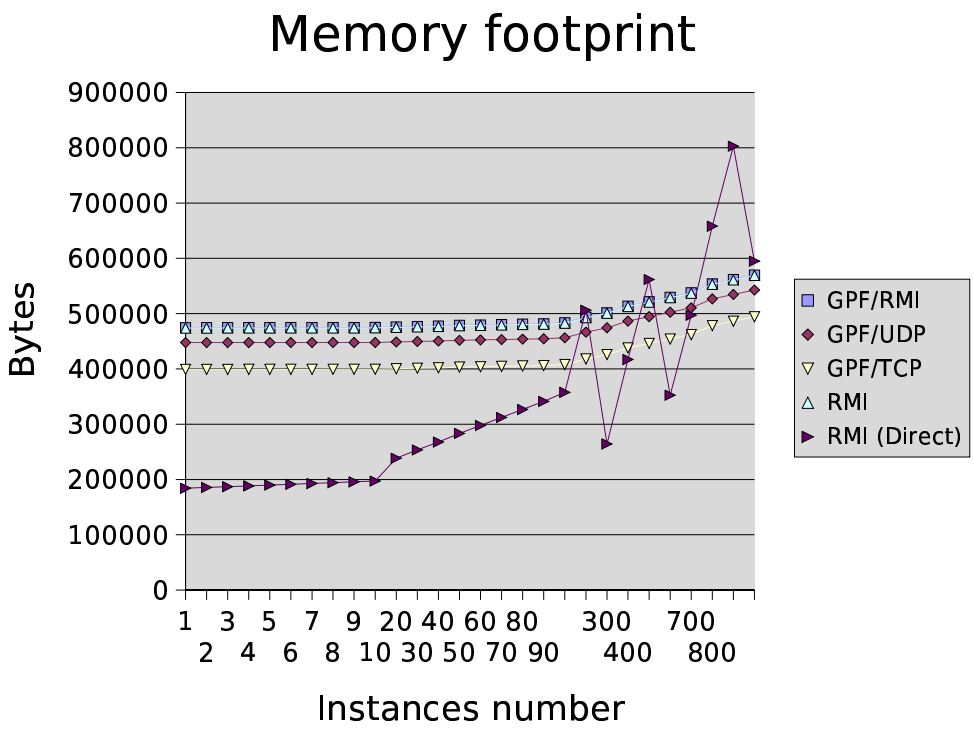


FIG. 19.6 – Consommation mémoire et objets distants.

200 Ko sont nécessaires à RMI pour la rendre distante. Le double est requis lorsqu'il s'agit d'un *stored object*. Dans ce cas toutefois, c'est le conteneur qui est rendu distant, pas le *stored object* : si ce dernier devient accessible par l'intermédiaire du conteneur, il ne fait l'objet d'aucun traitement particulier pour devenir distant. Cela tend à montrer que notre implémentation du concept des conteneurs actifs est relativement lourde. Elle utilise en effet massivement des *patterns* qui favorisent la maintenance, la réutilisation du code, au détriment de certaines optimisations en terme de complexité mémoire.

Par contre, cette figure montre que le coût de l'aspect distant est une fonction peu sensible au nombre d'instances, contrairement à RMI dont on peut noter l'irrégularité de l'empreinte mémoire dès la centaine d'instances dépassée. Enfin, on voit que l'architecture modulaire du GPF n'a pas d'impact significatif sur la complexité mémoire et que l'implémentation GPF/TCP est la moins gourmande des quatre implémentations de JACOb.

Nous avons aussi mesuré le temps nécessaire à l'obtention d'objets distants. La figure 19.7 présente les résultats. L'utilisation des conteneurs actifs a un coût qui n'est pas négligeable. Cependant, ce coût est là aussi une fonction peu sensible au nombre d'objets. L'avantage de JACOb est donc lié au dynamisme qu'il apporte : rendre un objet distant a un coût qui dépend seulement de la partie métier de l'objet

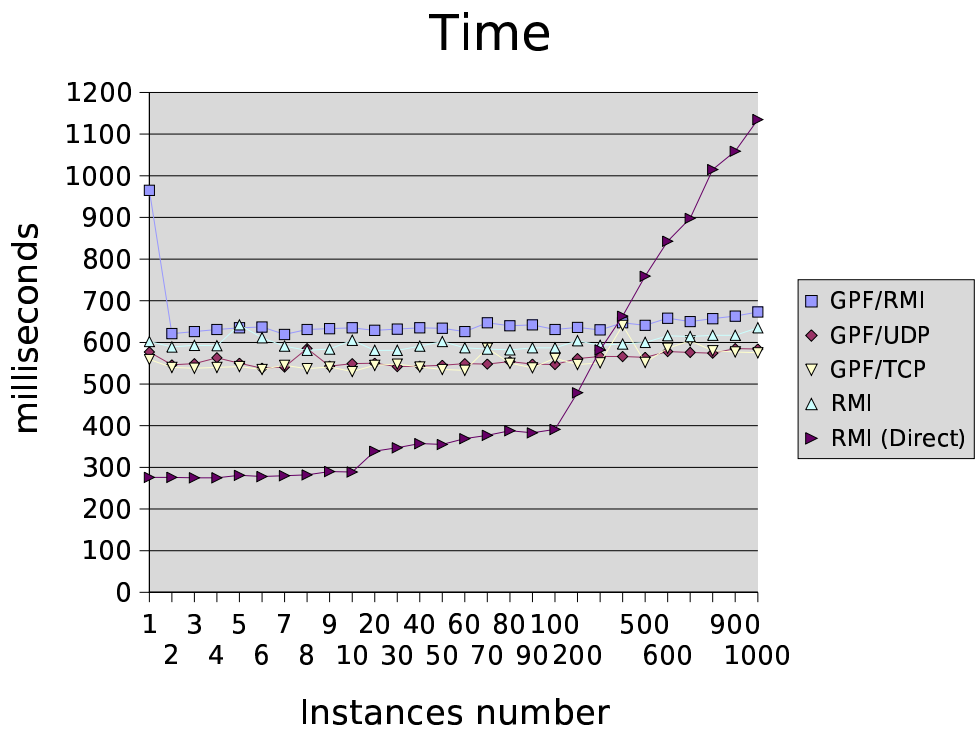


Fig. 19.7 – Temps requis pour rendre distant des objets.

et non d'un aspect distant, comme c'est le cas avec RMI.

Il faut garder à l'esprit que notre test utilise une classe métier vide. Dans des applications réelles, les objets distants sont souvent des objets lourds dont la logique métier possède une forte complexité. A ce titre, nos conteneurs sont de bons exemples d'objets distants lourds ; les figures précédentes montrent bien l'impact de leur implémentation sur la complexité en espace et en temps de leur instanciation.

De plus amples études mériteraient d'être réalisées pour étudier le comportement de notre implémentation face à des objets réels.

19.3 Bilan

Nos mesures de performances montrent qu'il est possible d'améliorer l'efficacité de certaines de nos classes. Nous pensons en particulier à notre implémentation du concept de *paire de références* qui semble être particulièrement inefficace, et à une implémentation suivant un modèle événementiel des deux couches de communications GPF/TCP et GPF/UDP.

Chapitre 20

Applications

Dans cette partie nous allons présenter quelques applications qui utilisent Mandala.

20.1 Agents mobiles

La première application écrite au-dessus de JACOb est une plate-forme d'agents mobiles minimaliste, très proche du modèle en π-calcul (*cf.* §D.1 p315). Elle a permis de soulever un certain nombre de problèmes liés au concept des conteneurs actifs.

20.1.1 Définition des entités de la plate-forme

Il est évidemment nécessaire de donner une définition claire des différentes entités qui constituent une plate-forme d'agents mobiles. Nous nous limiterons aux principales : l'*agent*, le *serveur d'agents* et le *proxy*. Ce dernier est utilisé par les clients qui veulent communiquer avec un agent. L'architecture utilisée est représentée sur la figure Fig. 20.1 p228. Le type de migration proposé est *faible* et *proactive* (*cf.* 3.1) en raison de la relative simplicité de l'implémentation et des limitations liées à l'utilisation d'une machine virtuelle Java (*cf.* 5.2.3).

Nous avons décidé de ne pas faire apparaître la notion de conteneur actif au niveau du client. Pour ce dernier, seules les trois entités de base sont visibles. Par conséquent, les *méthodes primitives* des conteneurs à savoir : `put()`, `get()`, `remove()` et `call()` ne sont pas accessibles. De fait, les communications inter-agents ne pourront avoir lieu directement en utilisant la méthode `call()`. Le *proxy* servira donc à la fois de référence distante, et d'intermédiaire pour la communication. L'API proposée aux clients (qui peuvent être également des agents) est rudimentaire. Deux méthodes sont disponibles :

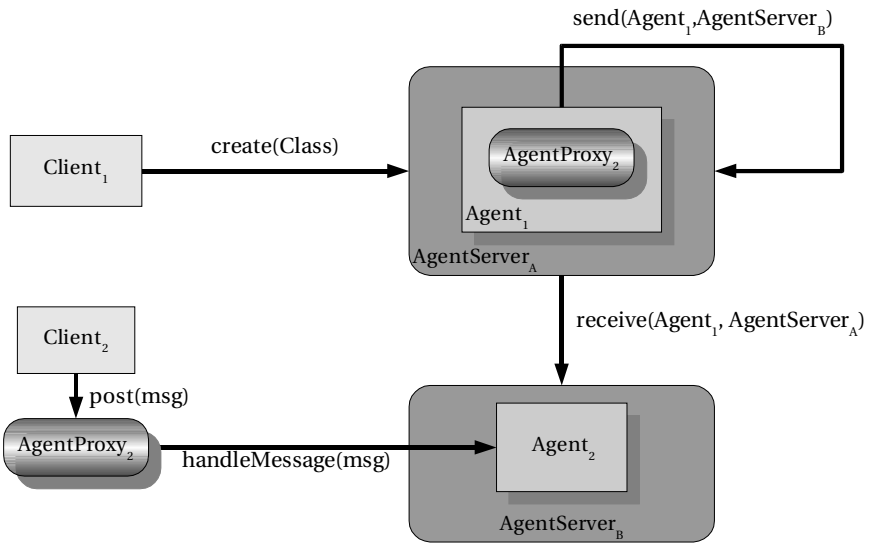


FIG. 20.1 – Architecture abstraite de notre système d'agents mobiles.

- **`AgentServer.create()` :** permet de créer un agent et qui retourne un *proxy* (la classe et les arguments du constructeur utilisés sont passés en paramètres) ;
- **`AgentProxy.post()` :** permet d'envoyer un message[1] à un agent.

Un agent est défini par une classe abstraite[2] dont il suffit d'hériter pour définir la logique métier de l'agent. Cette classe possède l'interface suivante :

```
1 public abstract class Agent implements Runnable, java.io.Serializable {
2     protected final void migrate(final AgentServer destination);
3     public void handleMessage(final Object msg);
4     public void onMigrating(final AgentServer destination);
5     public void onMigration(final AgentServer source);
6 }
```

Cette classe implémente l'interface `java.lang.Runnable`, sans définir la méthode `run()`. C'est dans cette dernière que la logique métier devra être implémentée. La méthode `migrate()` est déclarée `protected final` pour assurer une migration *proactive* : elle ne peut être invoquée par un objet d'une autre classe[3], et ne peut être redéfinie (et donc ne peut être rendue `public`). La méthode `handleMessage()` est invoquée chaque fois qu'un message est reçu par l'agent. On retrouve enfin les méthodes `onMigration()` et `onMigrating()` invoquées respectivement avant et après une migration[4].

[1] Un message est une instance de la classe `java.lang.String`.

[2] L'ensemble des classes de cet exemple est disponible dans le répertoire `Examples/src/agents/` de l'arborescence de Mandala [214].

[3] Mais elle peut être invoquée par un autre agent !

[4] Le lecteur aura reconnu l'interface des *aglets* que nous avons présentée en §4.3.2 p47.

Avec une telle API, nous proposons, comme exemple d'application, une version mobile du traditionnel `Hello World!` : le comportement de l'agent consiste à afficher régulièrement une chaîne de caractères. Le code métier d'un tel agent peut donc s'écrire[5] :

```
1 public void run() {
2     while(live) {
3         System.out.println("HelloWorld!");
4         Thread.sleep(1000);
5     }
6 }
```

L'agent dispose en outre d'une liste *circulaire* de serveurs à visiter. Chaque fois qu'il reçoit le message `MOVE`, il se déplace sur le serveur suivant. Par conséquent, le code de réception des messages peut s'écrire :

```
1 public void handleMessage(final Object msg) {
2     if (msg.equals(MOVE)) {
3         index++;
4         if (index >= serverList.length) index = 0;
5         migrate(serverList[index]);
6     }else System.err.println("Unknown command skipped: " + msg);
7 }
```

Enfin, pour assurer un comportement cohérent, nous devons nous assurer que l'agent s'arrête lorsqu'il a migré. En positionnant le champ `live` à `false` avant une migration, la boucle principale du code métier arrêtera l'agent. Par contre, l'agent, après migration, retrouvera son champ `live` à `false`, il convient de le positionner à `true`. D'où le code :

```
1 public void onMigrating(final AgentServer destination){
2     live = false;
3 }
4
5 public void onMigration(final AgentServer source){
6     live = true;
7 }
```

Notre agent est opérationnel. Une méthode `main()` dans la classe `HelloWorld-Agent` interprète les arguments de la ligne de commandes comme une liste de serveurs d'agents. Elle crée une instance de notre agent dans le premier d'entre eux et lance un `shell` minimal. Ce dernier lit sur l'entrée standard les messages à envoyer à l'agent

[5]La gestion des exceptions ne sera pas présentée tout au long de cet exemple dans un souci de concision. Le caractère purement démonstratif de celui-ci lui enlève de toute façon tout intérêt.

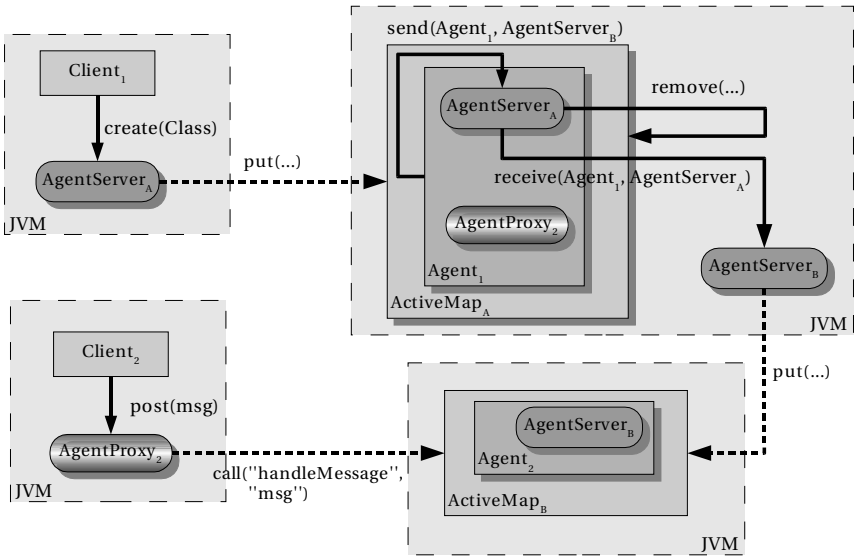


FIG. 20.2 – Architecture du système d'agents mobiles à base de conteneurs actifs.

par l'intermédiaire de son proxy. On note que, malgré les migrations de notre agent, les messages arrivent toujours à destination.

20.1.2 Mise en œuvre

Notre système d'agents mobiles doit être basé sur le concept des conteneurs actifs sans le faire apparaître au client. Par conséquent, ce sont les trois entités de base qui utiliseront les conteneurs. La figure FIG. 20.2 p230 présente l'architecture réelle retenue. Clairement, les entités de base sont des classes standard Java qui ne contiennent aucune logique particulière pour la gestion de l'aspect distant. Ce dernier sera fourni par les conteneurs.

20.1.2.1 La classe `Agent`

Nous allons commencer par décrire l'implémentation de l'agent qui est très simple. Celle en particulier de la méthode `migrate()` est déterminante pour le choix de l'architecture. Sur la figure, nos serveurs d'agents sont des instances de classe locale qui référencent des conteneurs actifs distants. Ils offrent aux clients la méthode `create()` qui permet de créer un agent ainsi que deux méthodes :

- `send(Agent agent, AgentServer destination)` ;
- `receive(Agent agent, AgentServer source)` ;

utilisées respectivement pour son envoi et sa réception. Ces deux dernières doivent avoir une visibilité limitée au paquetage afin d'éviter la séquence d'appels suivante depuis un client :

```
AgentServer source = ...;
AgentServer destination = ...;
Agent agent = ...;
destination.receive(agent, source);
```

Cela entraînerait les inconvénients suivants :

- la migration de l'agent ne serait plus *proactive*, mais *réactive* ;
- le serveur d'agent `source` ne serait pas informé de la migration (à moins que le serveur `destination` ne le signale).

Ce dernier inconvénient entraîne un problème de traçabilité, et surtout, un risque d'effectuer un clonage plutôt qu'une migration : le serveur `source` n'étant pas averti de la migration d'un de ses agents, il ne peut en supprimer[6] sa copie locale.

Par conséquent, nous avons choisi de placer l'ensemble de la logique de migration dans le code du serveur d'agents. La méthode `migrate()` de notre agent s'écrit donc :

```
public final void migrate(AgentServer destination) {
    hostedServer.send(this, destination);
}
```

Cela suppose que la classe `Agent` possède un champ `hostedServer`, qui représente le serveur d'agents courant. La mise à jour de ce champ sera réalisée par le serveur lui-même.

20.1.2.2 La classe AgentServer

La méthode `send()` du serveur consiste à invoquer la méthode `onMigrating()` de l'agent, à le cloner en demandant la réception au serveur destination, puis à le retirer du conteneur local :

```
void send(final Agent agent, final AgentServer destination) {
    agent.onMigrating(destination);
    destination.receive(agent, this);
    activeMap.remove(agent.key);
}
```

Ce code suppose que chaque serveur est associé à un conteneur actif, et chaque agent, à une clé lors de sa création.

L'ordre de la séquence d'appels est important : il assure que l'agent a bien été reçu avant de le retirer localement. Un mécanisme transactionnel devrait cependant

[6]En fait, déréférencer : c'est le ramasse-miettes qui libérera la mémoire allouée à l'agent en temps voulu.

être fourni pour garantir l'atomicité de l'opération consistant en l'appel séquentiel des méthodes `receive(), remove()`. Mais nous sortons du cadre de cet exemple.

La méthode `receive()` quant à elle, doit effectuer davantage d'opérations : l'agent passé en paramètre doit être inséré dans le conteneur, puis être initialisé. En particulier, le champ `hostedServer` doit être mis à jour. Une méthode `init()` est définie à cet effet dans la classe `Agent`. La méthode `onMigration()` doit être ensuite invoquée avant de démarrer une thread pour l'exécution du code métier de l'agent. Ainsi, une méthode `live()` est définie dans la classe `Agent`. L'utilisation d'une référence asynchrone distante sur l'agent – un *stored object* – facilite la mise en œuvre :

```
1  void receive(final Agent agent, final AgentServer source) {
2      activeMap.put(agent.key, agent);
3      final StoredObjectReference sor = getSOR(activeMap, agent.key);
4      sor.call(Agent.initMethod,
5               new Object[] {sor.getKey(), this}).waitUntilResultAvailable();
6      sor.call(Agent.onMigrationMethod,
7               new Object[]{source}).waitUntilResultAvailable();
8      sor.call(Agent.liveMethod, new Object[]{});
9  }
```

La méthode `getSOR()` retourne la référence asynchrone sur le *stored object* définie par le couple `(activeMap, key)` passé en paramètre. La gestion des communications est assurée par le conteneur qui contient l'agent. Par ailleurs, l'appel à la méthode `live()` se fait de manière asynchrone et distante, sans avoir à créer une *thread* à cet effet.

Celle permettant la création d'un agent est un peu plus complexe. Elle doit en effet :

- vérifier que la classe spécifiée est bien une classe dérivée de la classe `Agent` ;
- trouver le constructeur à utiliser en fonction des paramètres spécifiés ;
- instancier la classe à distance (le service générique d'instanciation à distance présenté en §10.6 p114 est utilisé à cet effet) ;
- intialiser l'agent (mettre à jour le champ `hostedServer` notamment) ;
- démarrer l'agent (en appelant la méthode `live()`).

Nous ne présenterons pas son implémentation. Notons cependant qu'il manque l'essentiel. Puisque cette méthode de création est destinée au client, elle doit lui retourner une information qui lui permet de référencer son agent et de communiquer avec lui. C'est la raison d'être du *proxy*, objet retourné par notre méthode `create()`.

20.1.2.3 Le *proxy* d'agent

Une instance de la classe `AgentProxy` permet à ses clients (qui peuvent aussi être des agents) de communiquer avec l'agent auquel elle est associée. Cette classe possède une seule méthode :

```
1 public void post(final Object message) {
2     sor.call(Agent.handleMessageMethod,
3                         new Object[]{message}).waitUntilResultAvailable();
4 }
```

Chaque *proxy* contient donc un champ `sor` qui est une référence sur un *stored object*. C'est par son intermédiaire que les communications ont lieu. La valeur de ce champ est déterminée à la création du *proxy* par la méthode `create()` de la classe `AgentServer` vue précédemment.

20.1.2.4 Communication et migration

Notre implémentation, si elle permet la migration des agents, n'assure pas complètement le transport des messages. En effet, nous avons vu que le *proxy* d'un agent maintient dans son champ `sor` la référence distante à l'agent. Cependant, après une migration, un agent n'est plus joignable puisque ce champ n'est pas mis à jour dans les *proxy*. Un mécanisme doit donc prévenir ces derniers que la référence de l'agent a changé :

1. lorsqu'un agent migre, il est remplacé par une entité, appelée *zombi*, qui assure la transmission des messages ;
2. lorsqu'un *zombi* est créé, la nouvelle référence distante de l'agent qu'il remplace lui est donnée ;
3. les agents et les *zombi* possèdent la même méthode

   `StoredObjectReference delegateMessage(String msg)`

   qui assure le traitement du message par la méthode `handleMessage()`, et qui retourne la référence de l'agent.

Le premier point implique la modification de la méthode `send()` de notre classe `AgentServer` :

```
1 void send(final Agent agent, final AgentServer destination) {
2     agent.onMigrating(destination);
3     final StoredObjectReference sor = destination.receive(agent, this);
4     // Replace the agent by a Zombi (instead of remove())
5     final Object o = activeMap.put(agent.sor.getKey(), new Zombi(sor));
6     agent.sor.call(Agent.initMethod,
```

```
7                   new Object[] {agent.sor,
8                                 this}).waitUntilResultAvailable();
9   }
```

Ce code suppose donc que l'agent dispose d'un champ `sor` qui est sa propre référence. Il remplace le champ `key` défini précédemment. La clé associée à l'agent est toujours disponible par l'appel `agent.sor.getKey()`. Ce champ est mis à jour lors de sa création et de sa migration par l'appel (à distance) de la méthode `init()`[7]. Dans le premier cas, c'est la méthode `create()` qui effectue l'appel, dans le second, c'est la méthode `receive()` :

```
1   StoredObjectReference receive(final Agent agent, final AgentServer source) {
2       // Remote call
3       activeMap.put(agent.sor.getKey(), agent);
4       final StoredObjectReference sor = getSOR(activeMap, agent.sor.getKey());
5       sor.call(Agent.initMethod,
6                new Object[] {sor, this}).waitUntilResultAvailable();
7       sor.call(Agent.onMigrationMethod,
8                new Object[]{source}).waitUntilResultAvailable();
9       // Asynchronous to prevent send() being blocked during removal.
10      sor.call(Agent.liveMethod, new Object[]{});
11      return sor;
12  }
```

Cette méthode `receive()` retourne la nouvelle référence de l'agent ce qui permet à la méthode `send()` de la passer en argument à la création du *zombi* de remplacement. Par ailleurs, il est important d'invoquer la méthode `live()` de l'agent de manière asynchrone. Si cette méthode était invoquée de manière synchrone, la méthode `receive()` retournerait la nouvelle référence à la méthode `send()` pendant l'exécution de l'agent (méthode `run()`). Ce dernier pourrait donc demander sa migration, et changer de référence avant que la méthode `send()` ait eu le temps de la mettre à jour dans son *zombi* de remplacement.

La classe `Zombi` est très simple :

```
1   public class Zombi extends Agent {
2       StoredObjectReference forward;
3       public Zombi(final StoredObjectReference forward) {
4           this.forward = forward;
5       }
6       public StoredObjectReference delegateMessage(final Object msg) {
7           final FutureClient f = forward.call(Agent.delegateMessageMethod,
8                                               new Object[]{msg});
```

[7]Dans la méthode `send()`, `init()` est invoquée sur le *zombi*, pas sur l'agent.

```
9           StoredObjectReference newForward = f.waitForResult();
10          if (newForward != forward) {
11                  // Agent moved!
12                  forward = newForward;
13          }
14          return forward;
15      }
16      public void run() {
17          throw new RuntimeException("Zombi can't live!");
18      }
19  }
```

La méthode `delegateMessage()` assure donc le transport des messages (par chaînage), quelque soit le serveur qui accueille l'agent. La méthode `delegateMessage()` de la classe `Agent` est triviale :

```
1  public StoredObjectReference delegateMessage(final Object msg) {
2      handleMessage(msg);
3      return sor;
4  }
```

Le proxy invoque donc la méthode `delegateMessage()` au lieu de `handleMessage()` pour acheminer les messages spécifiés dans sa méthode `post()` ; cela lui permet de savoir si la référence a changé :

```
1  public void post(final Object message) {
2      final FutureClient f = sor.call(Agent.delegateMessageMethod,
3                                      new Object[]{message});
4      final StoredObjectReference newSOR = f.waitForResult();
5      if (newSOR != sor) {
6          // New reference returned, the agent has moved!
7          sor = newSOR;
8      }
9  }
```

La logique interne des classes `Zombi` et `AgentProxy` est similaire : lorsque la référence distante a changé, l'agent a migré. La référence interne est mise à jour pour assurer que les prochains messages seront acheminés directement à l'agent. En évitant autant que possible[8] le chaînage des appels à la méthode `delegateMessage`, le coût du transport d'un message est diminué. Par conséquent, ce chaînage est temporaire : il est *brisé* dès le premier message envoyé. Les *zombis* peuvent donc persister dans

[8]Si un agent migre suffisamment souvent, il est possible qu'un message n'arrive pas à son agent destination.

le système. Un service générique (*cf.* §10.6 p114) implémentant un ramasse-miettes distribué permettrait d'éliminer les entités qui ne sont plus référencées[9] (*zombis* et agents).

Concernant le chaînage, le lecteur pourrait se demander pourquoi nous n'utilisons pas le chaînage de références présenté en §15.3 p172 : si ce mécanisme est une solution tout à fait envisageable, nous souhaitons présenter l'implémentation originale qui utilise seulement le concept des conteneurs actifs.

20.1.3 Communication inter-agents

Nous avons vu en §8.2.1 p76 qu'une migration supplémentaire est induite par l'utilisation directe des méthodes des conteneurs actifs. La migration d'un agent se résume à l'enchaînement des appels primitifs suivants :

```
Agent a = s1.get(key);
s1.remove(key);
s2.put(key, a);
```

Le lecteur pourra se référer à la figure FIG. 8.1 p77 pour se convaincre qu'une migration supplémentaire a bien lieu. On pourrait penser que nos deux appels `send()` et `receive()` ont le même problème[10]. En réalité, il n'en est rien. La migration étant *proactive*, un agent demande toujours sa migration en invoquant la méthode `migrate()`. Cette dernière utilise le champ `hostedServer` qui est local à l'agent, comme le montre la figure FIG. 20.2 p230. Un serveur d'agents possède une référence sur son conteneur actif associé. Si le serveur d'agents et le conteneur actif sont dans la même machine virtuelle, comme lors d'une demande de migration, les communications peuvent avoir lieu localement ; la migration supplémentaire est ainsi évitée.

20.2 Analyse des classes Java

A plusieurs reprises dans ce document, nous avons donné des statistiques sur les classes du JDK v1.4 de Sun (*cf.* §9.3 p82, §12.4.1.3 p139, §16.1.1 p180 et §16.1.3 p183). Nous avons écrit le code[11] qui produit ces statistiques : `ClassPathAnalyser`

[9] Les *stored objects* sont toujours référencés par leur conteneur. C'est ce qui permet de les rendre distants.

[10] Même si l'enchaînement des appels est différent pour les raisons évoquées en §20.1.2.2 p231

[11] Disponible dans le répertoire `Examples/src/analyser` de l'arborescence de Mandala [214].

est une version standard synchrone, et `RAMIClassPathAnalyser` la version asynchrone utilisant Mandala [12]. Nous avons utilisé la suite d'utilitaires LGPL de Cristiano Sadun [8]. Dans cette bibliothèque, on trouve la classe `SimpleClassPackageExplorer` qui permet d'explorer le contenu des paquetages Java.

Sans entrer dans les détails, l'objectif de notre application est d'analyser des classes Java afin de calculer un certain nombre de leurs caractéristiques (nombre de méthodes publiques, nombre de champs publics, nombre d'interfaces, etc.). Deux expressions régulières sont données en entrée pour filtrer les paquetages à analyser : un filtre d'exclusion comme premier argument et un filtre d'inclusion en second.

Par exemple, en spécifiant :

```
com.*|org.*     .*
```

toutes les classes n'appartenant pas à un *package* dont le nom commence par `com` ou par `org` sont analysés. Pour notre analyse des classes du JDK, nous avons utilisé les paramètres :

```
NONE     java.*
```

Ainsi, toutes les classes du JDK dont le nom commence par le préfixe `java` ont été prises en considération. Le code synchrone correspondant peut s'écrire[13] :

```
1  public Data explore() {
2      final Data data = new Data();
3      final String[] packageNames = explorer.listPackageNames();
4      nextPackage:
5      for(int i = packageNames.length - 1; i >= 0; i--) {
6          // Filtering
7          if (exclude.matcher(packageNames[i]).matches()) {
8              log.config(packageNames[i] + " filtered (in excludePattern)");
9              continue nextPackage;
10         }
11         if (!include.matcher(packageNames[i]).matches()) {
12             log.config(packageNames[i] + " filtered (not in includePattern)");
13             continue nextPackage;
14         }
15         log.fine("Exploring " + packageNames[i]);
16         final String[] classes = explorer.listPackage(packageNames[i]);
17         data.classes += classes.length;
18         for (int j = classes.length - 1; j >= 0; j--) {
19             try{
20                 log.finer("Analysing class: " + classes[j]);
```

[12] L'aspect distant n'ayant que peu d'intérêt dans cet exemple, c'est RAMI seulement qui est utilisé.

[13] Nous présentons volontairement la gestion des exceptions dans les extraits afin de la comparer avec la version asynchrone.

```
21                  analyseClass(Class.forName(classes[j]), data);
22              }catch(Throwable t) {
23                  log.info("Throwable thrown, skipping: " +
24                          classes[j] + "(Throwable: " +
25                          t.getMessage() + ")");
26              }
27          }
28      }
29      return data;
30  }
```

La méthode `explore()` retourne les résultats de l'analyse dans un objet de type `Data`. Après le filtrage, la liste des classes est récupérée en utilisant la méthode `listPackage()` de l'objet `explorer`. Celui-ci est une instance de la classe `SimpleClassPackageExplorer` fournie par la bibliothèque d'utilitaires de Cristiano Sadun. Pour chaque classe de cette liste, l'analyse est réalisée en utilisant une méthode (`analyseClass()`) que nous ne décrirons pas. Clairement, il apparaît que l'analyse des classes peut être effectuée indépendamment de l'exploration des paquetages. L'utilisation de RAMI permet d'écrire :

```
1   public Data explore() {
2       final String[] packageNames = explorer.listPackageNames();
3       final Data data = new Data();
4       FutureClient future = null;
5       final ClassesCallback callback = new ClassesCallback(data);
6       nextPackage:
7       for(int i = packageNames.length - 1; i >= 0; i--) {
8           // Filtering
9           if (exclude.matcher(packageNames[i]).matches()) {
10              log.config(packageNames[i] + " filtered (in excludePattern)");
11              continue nextPackage;
12          }
13          if (!include.matcher(packageNames[i]).matches()) {
14              log.config(packageNames[i] + " filtered (not in includePattern)");
15              continue nextPackage;
16          }
17          log.fine("Exploring " + packageNames[i] + " asynchronously");
18          future = explorerProxy.rami_listPackage(packageNames[i], callback);
19      }
20
21      // TERMINATION TEST
22  }
```

Le principe est donc d'invoquer la méthode `explorer.listPackage()` de manière asynchrone afin de rendre l'analyse des classes concurrente de l'exploration

des paquetages et du filtrage. Elle peut se faire au plus tôt en utilisant un *callback* (*cf.* §15.1.2 p160) dont le code est le suivant :

```
1  private class ClassesCallback implements Callback {
2      final Data data;
3      ClassesCallback(final Data data) {
4          this.data = data;
5      }
6      public void done(final InvocationInfo info, final MethodResult result) {
7          final String[] classes;
8          try{
9              final String[] classes = (String[]) result.getReturnedResult();
10         }catch(Throwable t) {
11             log.log(Level.SEVERE,
12                     "Exception during an asynchronous method invocation",
13                     t);
14             return;
15         }
16         data.classes += classes.length;
17         for (int i = classes.length - 1; i >= 0; i--) {
18             try{
19                 log.finer("Analysing class: " + classes[i]);
20                 analyseClass(Class.forName(classes[i]), data);
21             }catch(Throwable t) {
22                 log.info("Throwable thrown, skipping: " +
23                          classes[i] + "(Throwable: " +
24                          t.getMessage() + ")");
25             }
26         }
27     }
28 }
```

Pour chaque chaîne de caractères de la liste retournée par l'appel asynchrone de la méthode `explorer.listPackage()`, la classe correspondante est retrouvée (en utilisant `Class.forName()`). L'analyse est réalisée en invoquant (de manière synchrone) la méthode `analyseClass()`. Nous devons également tenir compte du problème de terminaison : si l'exploration des paquetages se termine avant les analyses, il faut que la *thread* principale attende. Nous aurions pu utiliser à cet effet le *callback* `ResultsGrouper` qui permet d'attendre la terminaison de tous les appels (*cf.* §14.7 p155), mais nous verrons bientôt une autre alternative.

Lorsque nous utilisons un paradigme d'appel de méthode asynchrone, il faut généralement spécifier la sémantique asynchrone utilisée. Dans le cas de RAMI, les politiques asynchrones sont créées par les implémentations de l'interface `AsynchronousPolicyFactory` présentée brièvement en §18.3 p202. Lorsque rien n'est

spécifié, l'instance retournée par la méthode `getDefaultAPFactory()` de la classe `mandala.rami.impl.ARFactory` est utilisée pour la création des politiques. Elle est paramétrée par la *property* `mandala.rami.impl.apf` qui est associée à la classe `mandala.rami.impl.ThreadPooledAPFactory` par défaut[14]. Plutôt que d'utiliser une politique par défaut, la prudence impose d'en utiliser une non-concurrente pour les instances dont la classe ne précise pas clairement sa tolérance aux accès concurrents. Dans notre cas, le code source de la classe `SimpleClassPackageExplorer` est disponible. À l'évidence, il n'est pas prévu pour être utilisé dans un cadre *multi-threadé*. Aussi, une politique `FIFO`[15] est spécifiée lors de la récupération du proxy asynchrone semi-transparent (*cf.* §16.2 p186) :

```
1 final Framework.Factory factory = new ARFactory(new FifoAPFactory());
2 this.explorerProxy = (jaya.org.sadun.util.ClassPackageExplorer)
3         Framework.getSemiTransparentAsynchronousProxy(explorer, factory);
```

La politique `FIFO` garantit que tous les appels précédents sont finis lorsque le dernier appel à `FutureClient.waitUntilResultAvailable()` se termine. Notre test de terminaison peut donc s'écrire :

```
1     // Fifo implies the last method is the last invoked!
2     try{
3         if (future != null) future.waitUntilResultAvailable();
4     }catch(InterruptedException ie) {
5         log.log(Level.WARNING,
6                 "Interrupted during waitUntilResultAvailable()",
7                 ie);
8     }
9     return data;
```

Nous pourrions diminuer encore la granularité de la concurrence : la méthode `analyseClass()` fait appel pour chaque méthode d'une classe à une méthode `analyseMethod()` chargée de l'analyser. Cette opération pourrait également se faire en concurrence. En utilisant un proxy asynchrone sur les instances de la classe réflexive `java.lang.Class`, la méthode `Class.getDeclaredMethod()` pourrait être invoquée de manière asynchrone. Cela permettrait de réaliser (à l'aide d'un *callback* approprié) les analyses de méthodes en concurrence de l'exploration des classes et des paquetages. La même approche peut aussi être utilisée pour les champs (méthode `Class.getDeclaredFields()`). Toutefois, plus la granularité est fine, plus le coût de gestion de la concurrence est important. Trouver la bonne granularité requiert une étude approfondie de l'application. Dans cet exemple, les analyses (de classes,

[14]Le lecteur peut se référer à la *Javadoc* en ligne [212] pour de plus amples informations.

[15]Une politique `Random` n'aurait aucun intérêt.

de méthodes et de champs) sont d'une complexité trop faible pour qu'un gain soit tiré d'une granularité plus fine (à moins d'exécuter le programme sur un très grand nombre de paquetages).

20.3 Calcul parallèle du nombre π

En reprenant l'exemple du calcul des n premières décimales du nombre π utilisé pour nos mesures de performances en §19.1.2 p213 et §19.2.2 p220, nous nous sommes intéressés à la parallélisation de l'algorithme.

Dans un contexte homogène dédié (*cluster* par exemple), l'algorithme est trivial : on utilise le découpage en p sous-sommes décrit en §19.1.2 p214 où p est égal au nombre de processeurs disponibles.

20.3.1 Algorithme sur un réseau de machines hétérogènes *dédiées*

Dans un cadre hétérogène, on dispose de p machines de performances différentes. On suppose que l'on est capable d'évaluer leurs performances relatives et de les trier dans un ordre croissant : $\lambda_1 < \ldots < \lambda_p$. Pour une somme $s = \sum_{i=\alpha}^{\beta} f(i)$ donnée, on note $|s|$ sa *taille* définie par $|s| = \beta - \alpha + 1$. La complexité temporelle C du calcul d'une somme s sur une machine dont la performance relative est évaluée à λ est alors donnée par l'expression : $C = \lambda.|s|$.

Pour calculer les n premières décimales de π, on cherche donc une décomposition en p sous-sommes $s_1, \ldots, s_p$ du calcul S_n original et à assigner chaque sous-somme à une machine de telle sorte que :

1. les sous-sommes s_k soient indépendantes les unes des autres :

$$s_k = \sum_{i=b_k}^{e_k} f(i), \text{ tel que } \forall k, l \in [\![1, p]\!], k \neq l, [\![b_k, e_k]\!] \cap [\![b_l, e_l]\!] = \emptyset$$

2. la somme des p sous-sommes calcule bien les n premières décimales de π :

$$S_n = \sum_{k=1}^{p} (s_k) = \sum_{i=0}^{n} f(i)$$

3. la taille des sous-sommes soit adaptée aux puissances relatives afin d'optimiser l'utilisation des ressources processeurs disponibles dans les machines : le temps d'inactivité des machines doit être réduit au minimum ; on appele *divergence* et

on note Δ la différence entre la complexité maximale – atteinte par la machine m – et les autres complexités :

$$\Delta = \sum_{k \neq m} C_m - C_k = \lambda_m.|s_m| - \lambda_k.|s_k|, \quad C_m = max\{C_k, k \in [\![1, p]\!]\}$$

4. la solution soit optimale, c'est à dire que sa complexité temporelle doit être minimale.

En considérant que la somme de deux nombres x et y de même ordre de grandeur (même nombre de décimales) est une opération constante, un algorithme[16] pour la résolution d'un tel système est relativement trivial : l'idée est d'augmenter de 1 la taille d'une sous-somme assignée à une machine si c'est l'opération qui améliore le plus la divergence Δ :

```
|s_1| = 1      // On commence par augmenter la taille
C_1 = λ_0.|s_1| // sur la machine la plus performante
for (i = 0; i ≤ n; i = i + 1) {      // Somme de 0 à n !
    m = 1      // Machine candidate à l'augmentation
    Δ = ∞      // Plus petite divergence courante
    for (k = 1; k ≤ p; k = k + 1) {
        Γ = λ_k.(|s_k| + 1)      // Complexité future
        δ = Σ_{l≠k} Γ − C_l      // Divergence future
        if (δ < Δ) {
            // En augmentant de 1 la taille de s_k
            // on obtient une meilleure divergence.
            // La machine k est donc sélectionnée
            m = k,   Δ = δ
        }
    }
    |s_m| = |s_m| + 1,   C_m = λ_m.|s_m|
}
```

Schématiquement, on représente les machines verticalement sur un axe des temps comme sur la figure FIG. 20.3 p244. L'unité de temps est égale à la performance relative de la première machine (qui peut toujours être ramené à 1 par division). Les augmentations de taille des sous-sommes sont représentées par des blocs dont la longueur est proportionnelle à la performance de la machine à laquelle ils sont assignés. Ainsi, une augmentation de 1 de la taille d'une sous-somme assignée à une machine de performance relative égale à 2 est représentée par un bloc de longueur

[16] On trouvera le code correspondant à cet algorithme dans la classe `Dispatcher` du répertoire `Examples/src/pi` de l'arborescence de Mandala.

égale à 2 unités de temps. Au départ, il y a $n-1$ sommes à effectuer, et p sous-sommes, $s_1, \ldots, s_p$, assignées toutes de taille nulle sauf la première ($|s_1| = 1, |s_k| = 0, 1 < k \leq p$).

Le déroulement de l'algorithme sur un petit exemple est illustré sur la figure : on cherche à calculer les $n = 9$ premières décimales du nombre π à l'aide de 3 machines `M1`, `M2`, et `M3` de performances relatives respectives 1, 2 et 3. A chaque étape (il y en a $n+1 = 10$), on cherche l'augmentation de la taille d'une sous-somme qui diminuera le plus Δ. On calcule donc Δ pour chaque cas. Pour la première étape où $|s_0|$ vaut déjà 1, on a les trois cas de figure suivants :

1. si s_1 est augmenté, $\Delta = (1 * 2 - 2 * 0) + (1 * 2) - (3 * 0) = 4$;
2. si s_2 est augmenté, $\Delta = (2 * 1 - 1 * 1) + (2 * 1 - 3 * 0) = 3$;
3. si s_3 est augmenté, $\Delta = (3 * 1 - 1 * 1) + (3 * 1 - 2 * 0) = 5$.

Par conséquent, c'est s_2 qui est augmenté. En réitérant le processus, on obtient la solution :

$$|s_1| = 5, |s_2| = 3, |s_3| = 2, \quad \Delta = 1$$

Notons que

$$|s_1| = 6, |s_2| = 2, |s_3| = 2$$

est aussi une solution en 6 unités de temps (et il y en a d'autres...), mais elle ne vérifie pas le critère 3 : en effet la divergence vaut dans ce cas : $\Delta = 2 + 0 = 2$ qui est supérieure à celle de la solution de notre algorithme. Par ailleurs, le choix des sous-sommes n'a pas d'importance, c'est leur taille qui joue sur la complexité du calcul. On choisira arbitrairement les sous-sommes suivantes :

$$s_1 = \sum_{k=0}^{4} f(k), s_2 = \sum_{k=5}^{7} f(k), s3 = \sum_{k=8}^{9} f(k)$$

L'inconvénient de cet algorithme est lié à la nécessité de connaître précisément les performances relatives de chaque machine utilisée. Dans le cas général, il n'est pas facile d'effectuer de telles mesures : de nombreux paramètres entrent en jeu et restent difficilement quantifiables (paramètre réseau en particulier, mais aussi interactions avec le système, cache processeur, modèle mémoire, etc.). En Java, ces évaluations sont encore plus compliquées à réaliser car la machine virtuelle contient de nombreux paramètres, et les interventions sporadiques du ramasse-miettes perturbent les mesures.

Enfin, nous nous sommes aussi intéressé aux problèmes de tolérance aux pannes ce qui nous a amené à implémenter un algorithme d'équilibrage de charge dynamique tolérant aux pannes sur un réseau de machines hétérogènes *non-dédiées* : des machines de bureau seront utilisées pour réaliser le calcul.

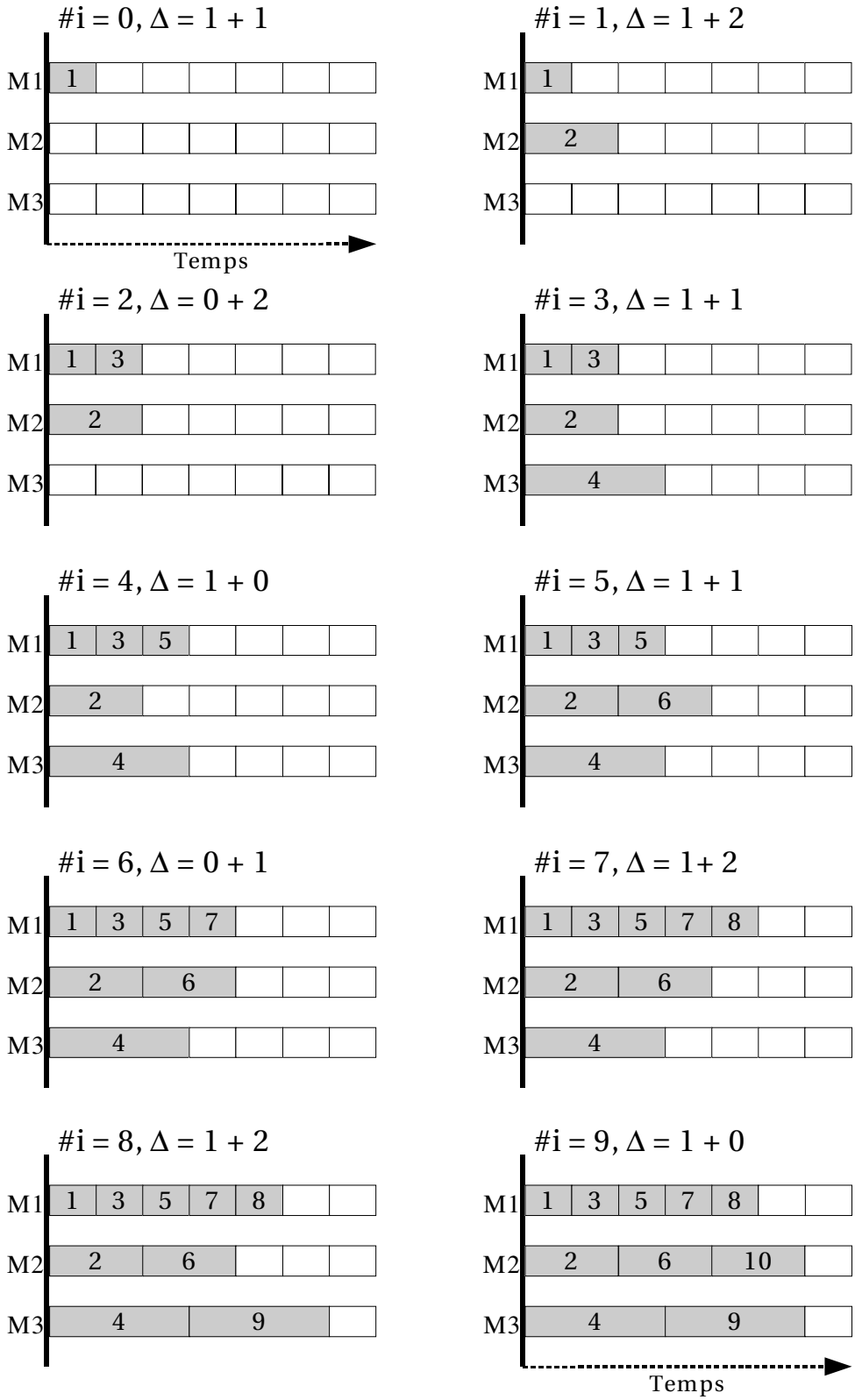


FIG. 20.3 – Illustration de l'algorithme du calcul de π sur un réseau de machines hétérogènes *dédiées*.

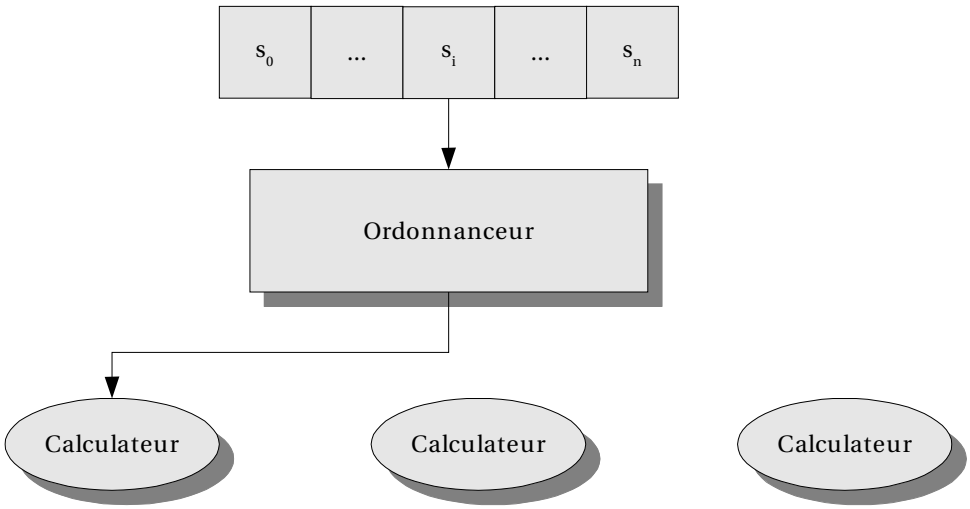


Fig. 20.4 – Représentation abstraite du calcul de π sur un réseau de machines hétérogènes *non-dédiées*.

20.3.2 Algorithme sur réseau de machines hétérogènes *non-dédiées*

Il existe de nombreux algorithmes pour le calcul du nombre π [3], mais nous allons réutiliser la classe `PiComputer` qui nous a servi à réaliser des études de performances (*cf.* §19.1.2 p213 et §19.2.2 p220) ; cela nous donne en même temps l'occasion de montrer l'intérêt de l'aspect *dynamique* de Mandala.

Le principe est de déployer des *calculateurs* – en fait des instances de la classe `PiComputer` – sur chaque machine et de leur donner du travail : des sous-sommes à calculer. L'ordonnanceur doit assurer :

- que tout le monde travaille ;
- que les machines qui ne répondent pas (ou plus) ne bloquent pas l'ensemble du calcul.

A cet effet, on dispose d'un réservoir de sous-sommes à calculer qui sont envoyées à un calculateur chaque fois qu'il le demande comme le représente la figure Fig. 20.4 p245. Lorsque toutes les sous-sommes ont été envoyées mais pas encore calculées, un algorithme est mis en place pour permettre aux machines qui ont terminé, de calculer des sous-sommes déjà assignées. Lorsqu'un calculateur a terminé le calcul d'une telle sous-somme, il interrompt les autres calculateurs qui ont aussi cette sous-somme d'assignée. Le nombre de sous-sommes est paramétré au départ par l'utilisateur.

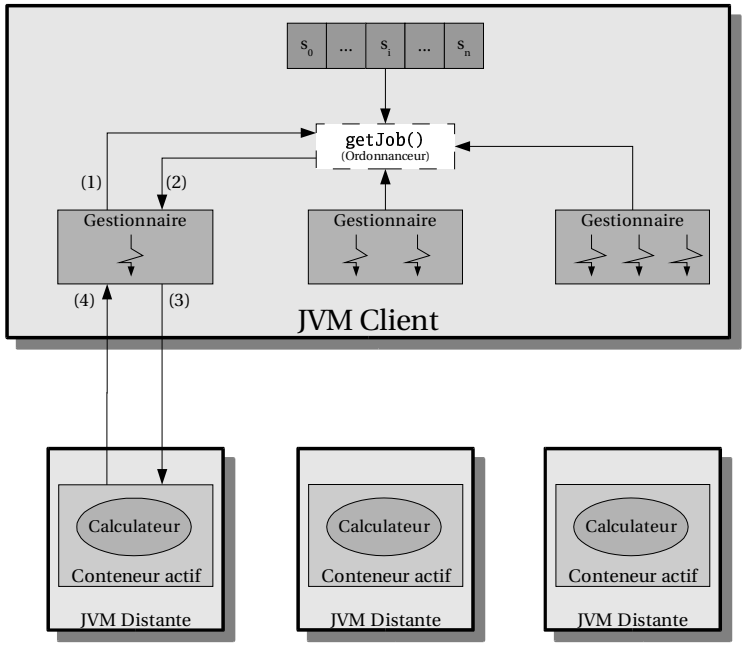


FIG. 20.5 – Architecture utilisée pour le calcul de π sur un réseau de machines hétérogènes *non-dédiées*.

20.3.3 Mise en œuvre

L'utilisation de Mandala facilite l'implémentation d'un tel programme. La figure FIG. 20.5 p246 présente les différentes entités de l'architecture implémentée.

Les calculateurs sont des instances de la classe `PiComputer` qui sont insérés dans des conteneurs actifs faisant ainsi des *stored objects* (*cf.* §9.11.1 p91). Des proxy asynchrones semi-transparents (*cf.* §16.2 p186) sur ces objets sont utilisés. Ces proxy sont liés à des références asynchrones de type `StoredObjectReference` (*cf.* §10.4 p111).

20.3.3.1 Insertions asynchrones des calculateurs

Le dynamisme de Mandala permet d'implémenter assez facilement un mécanisme d'insertions asynchrones qui optimise le démarrage de l'application : il est en effet inutile d'attendre que tous les calculateurs soient "prêts" (insérés), le calcul peut démarrer dès la première insertion. A cet effet, nous avons utilisé le service générique d'instanciation à distance (*cf.* §10.6 p114) fourni par la classe `Instanciator` brièvement présentée en §2.2.4 p15. Cette classe contient la méthode suivante :

```
1 // Returns the key mapped to the freshly instanciated stored object
2 public Object instanciate(final Object[] args);
```

Celle-ci :

- instancie une classe à l'aide d'un constructeur par défaut (spécifié à l'instanciation de la classe `Instanciator`) ;
- lui passe en paramètre les arguments spécifiés par le tableau `args` ;
- insére l'objet correspondant dans le conteneur auquel il est associé (spécifié à l'instanciation) ;
- et retourne la clé attachée qu'il aura pris soin de créer globalement unique.

Le code d'instanciations asynchrones faisant appel à ce service est le suivant[17] :

```
1  final jaya.mandala.jacob.Instanciator[] instanciators =
2              new jaya.mandala.jacob.Instanciator[activeMaps.length];
3
4  // Instanciate asynchronously and remotely PiComputers into active maps
5  for (int i = 0; i < activeMaps.length; i++) {
6      StoredObjectReference sor = Instanciator.getInstance(activeMaps[i],
7                                                           piComputerConstructor);
8      instanciators[i] = (jaya.mandala.jacob.Instanciator)
9                  jaya.mandala.jacob.Instanciator.getInstance(sor);
10     instanciators[i].rami_instanciate(new Object[]{scale},
11                                       new RemoteNewCallback(i));
12 }
```

Comme on le voit, la méthode `instanciate()` est appelée de manière asynchrone avec pour argument le nombre de décimales souhaité par l'utilisateur (`scale`). Le retour de cet appel, un *future* (*cf.* §15.1.1 p158) n'est pas utilisé. En effet, le résultat est récupéré par un *callback* (*cf.* §15.1.2 p160) ce qui permet de démarrer le calcul au plus tôt. Ce dernier a pour charge de créer un gestionnaire de calculateur (`PiManager`).

20.3.3.2 Gestionnaires de calculateur

Chaque gestionnaire demande un *job* (étape (1) sur la figure) par l'intérmédiaire de la méthode `getJob()` qui retourne `null` lorsqu'il peut arrêter de travailler (ainsi que son calculateur associé). Un *job* est défini par :

- **une sous-somme ;**
- **un état :** assigné, non assigné, ou calculé ;
- **une liste de `FutureClient`** qui permet de trouver les éventuels autres calculateurs qui effectuent la sous-somme correspondante ;
- **deux tableaux de date d'assignation et de retour** utilisés par l'algorithme d'équilibrage de charge et de tolérance aux pannes.

[17] L'essentiel du code de cette application se trouve dans la classe `PiDistributedComputer.java` du répertoire `Examples/src/pi/` de l'arborescence de Mandala [214].

Lorsqu'un *job* est récupéré par un gestionnaire (étape (2)), la sous-somme associée est envoyée à un calculateur (étape (3)) en invoquant la méthode `PiComputer.compute()` de manière distante et asynchrone. Le *future* retourné par cet appel est placé dans la liste correspondante du *job*. Lorsque le calcul est terminé (étape (4)), un certain nombre de statistiques sont calculées localement par le gestionnaire : temps de calcul réel, temps de l'appel distant, coût de la communication ; elles seront utiles pour l'équilibrage de charge. Enfin, un *job* pouvant être assigné à plusieurs calculateurs, il faut informer les autres gestionnaires que le calcul est terminé. Les mécanismes d'annulations (§14.5 p153) sont utilisés à cet effet. Par conséquent, le code du gestionnaire est le suivant :

```
1  while (true) {
2      try{
3          synchronized(lock) {
4              job = getJob(job);
5              if (job == null) {
6                  computeStats(...);
7                  stopWorker();
8                  return;
9              }
10             job.selectedDates[rank] = System.currentTimeMillis();
11         }
12         // sending phase
13         job.futures[rank] = computers[rank].rami_compute(job.start, job.end);
14         // computing little local stats
15         ...
16         // waiting phase
17         PiComputer.Result result = job.futures[rank].waitForResult();
18         // computing local stats
19         job.returnedDates[rank] = System.currentTimeMillis();
20         ...
21         // Inform others that this job has been computed
22         synchronized(job) {
23             if (Thread.interrupted())
24                 throw new InterruptedException("Interrupted in the " +
25                                                "synchronized section !");
26             }
27             // Only the thread which has computed job and which get the job
28             // lock first can be in this synchronized section.
29
30             // Inform that this job has been computed. See reselect()
31             job.state = Job.COMPUTED;
32             // Interrupt others
33             ...
34         }
35         // Use result HERE and not before (concurrent
```

```
36          // threads may use two times the same result!)
37          ...
38      }catch(Throwable e) {
39          // handle exception
40      }
```

Chaque gestionnaire possède un certain nombre de threads (paramétré par l'utilisateur) qui exécutent cette boucle. Les calculateurs peuvent en effet être sur des machines multi-processeurs, il peut donc être intéressant de leur assigner plusieurs sous-sommes en même temps. Le moniteur `lock` est donc commun à toutes les threads de l'application. Chaque thread tente de récupérer un *job* en utilisant la méthode `getJob()`. C'est donc cette méthode qui assigne les sous-sommes aux calculateurs, elle implémente donc l'algorithme d'équilibrage de charge.

C'est cet algorithme que nous allons présenter.

20.3.3.3 Équilibrage de charge

Deux listes sont maintenues : la liste des *jobs* à calculer, `Job[] jobs`, et la liste des *jobs* en cours de calculs, `computingJobs`. Lorsque la méthode `getJob()` est invoquée, un *job* lui est passé en paramètre correspondant à la sous-somme précédemment calculée. Il convient donc de la retirer de la liste `computingJobs`. Ce paramètre peut être `null` si le calcul n'a pu être réalisé (une exception a été levée). Ensuite, les sous-sommes sont assignées une à une séquentiellement tant qu'il en reste à calculer. D'où le code :

```
1  // Must be synchronized on lock
2  private Job getJob(final Job previous) {
3      if (previous != null) {
4          // previous has been computed by this currentThread.
5          computingJobs.remove(previous);
6          if (currentSubSum < subsums.length) {
7              final int packetToSend = currentSubSum++;
8              firstSelect(jobs[packetToSend], packetToSend);
9              return jobs[packetToSend];
10         }
```

la méthode `firstSelect()` s'occupe de préparer le *job* : positionner l'état à `Job.ASSIGNED` essentiellement. Dans le cas où tous les *jobs* ont été assignés, *i.e.*

```
currentSubSum >= subsums.length
```

, il faut vérifier qu'il reste des *jobs* à calculer :

```
1  if (computingJobs.isEmpty()) return null;
```

sinon, l'équilibrage de charge commence. A ce stade, la thread qui invoque `getJob()` vient de terminer le calcul d'une sous-somme et en demande une nouvelle. Tous les *jobs* sont assignés, mais certains ne sont pas encore calculés. L'idée est d'assigner un *job* à la thread courante si son calcul prendra moins de temps que le temps restant. Il faut donc estimer :

- le temps moyen de calcul d'une sous-somme par le calculateur associé à la thread courante (qui appartient à un gestionnaire donné) ;
- le temps restant pour un *job* donné.

Nous avons vu que chaque thread d'un gestionnaire établit des moyennes sur les temps de calculs. Aussi, on parcourt la liste `computingJobs` à la recherche d'un job que la thread courante peut terminer en moins de temps que le temps restant. La méthode a accès au champ `averageTime` de chaque gestionnaire qui représente le temps moyen de calcul d'une sous-somme pour le calculateur associé. Les gestionnaires sont stockés dans un tableau ce qui permet à la méthode `getJob()` d'y accéder par l'intermédiaire d'un simple indice. Par ailleurs, chaque gestionnaire connaît son propre indice : `rank`. Durant le parcours, on prévoit à l'avance le cas où aucun *job* ne correspond : le calculateur associé à la thread courante possède un `averageTime` trop important pour permettre d'améliorer le temps de calcul d'un des *jobs* de la liste `computingJobs`. On sélectionne dans ce cas le premier job. Nous verrons que cela répond à une tolérance aux pannes.

```
1  final Iterator iterator = computingJobs.iterator();
2  final Job faultToleranceSelection = null;
3  while(iterator.hasNext()) {
4      final Job job = (Job) iterator.next();
5      // Test if this job has already been assigned to managers[rank]!
6      if (job.selectedDates[rank] != Job.UNSELECTED) continue;
7
8      // Fault tolerance algorithm select the first job in the list
9      if (faultToleranceSelection == null) faultToleranceSelection = job;
10
11     float minRemainingTime = computeMinRemainingTime(job);
12
13     // Test if this manager averageTime is less than
14     // the minimum remaining computing time
15     if (minRemainingTime > averageTime) {
16         if (reSelect(job)) return job;
17         else continue;
18     }
19
20     // Always select a job even if you can't do better (fault
21     // tolerance algorithm)
22     if (faultToleranceSelection == null) return null;
23     if (reSelect(faultToleranceSelection)) return faultToleranceSelection;
```

```
24      return null;
25  }
```

La méthode `reSelect()` garantit que la sélection du *job* n'intervient pas au moment même où il est terminé : la méthode `getJob()` est en effet vérouillée sur l'objet global `lock`, mais une fois qu'une thread d'un gestionnaire a reçu un job, elle sort du bloc `synchronized(lock)` et transfert la sous-somme au calculateur associé. Par conséquent, un *job* peut être resélectionné alors qu'il est en train de se terminer. La thread qui a réalisé la demande rentre alors dans le bloc `synchronised(job)` chargé d'informer les autres que le *job* est dans l'état `Job.COMPUTED`. La méthode `reSelect()` se synchronise donc sur l'objet `job` avant de l'assigner ce qui garantit que :

- si le *job* est passé à l'état `Job.COMPUTED`, il ne sera pas sélectionné (`reSelect()` retournera `false`) ;
- si le *job* est finalement sélectionné (`reSelect()` retourne `true`), l'ensemble des informations permettant l'annulation de la requête seront mises à jour.

L'algorithme de tolérance aux pannes est très simple : si une exception est levée dans le corps du gestionnaire (la boucle `while()`), la requête est annulée (dans la mesure du possible), et on redemande un job en faisant appel à la méthode `getJob()`. Un paramètre `null` est passé pour éviter que le *job* qui n'a pas été calculé (puisqu'il y a eu une exception) soit retiré de la liste `computingJobs`.

20.3.3.4 Tri de la liste de *jobs* en cours de calcul

La liste `computingJobs` est triée[18]. L'ordre a en effet une importance :

- l'algorithme d'équilibrage de charge choisit le premier *job* qui a une chance d'être calculé plus rapidement par la thread courante ;
- l'algorithme de tolérance aux pannes sélectionne systématiquement le premier *job* de cette liste.

Si la liste n'était pas triée, il serait possible d'avoir un *job* calculé par k calculateurs en même temps, et d'autres *jobs* calculés par un 1 seul. Pour assurer un bon équilibre, lorsqu'un *job* est assigné à un calculateur, il est réordonné dans la liste. Le critère de comparaison est le suivant :

> *si deux* jobs *ont le même nombre d'assignation, le plus vieux et prioritaire, sinon, celui qui a le moins d'assignation est prioritaire.*

Le premier critère (le plus vieux est prioritaire) favorise la tolérance aux pannes : un *job* qui n'a pas été calculé depuis longtemps sera le premier sélectionné, ce qui

[18] C'est une instance de la classe `java.util.TreeSet`.

évite de pénaliser l'ensemble du calcul, surtout pour des *jobs* dont le temps de calcul est long.

Le second critère (celui qui a le moins d'assignation est prioritaire) favorise l'équilibrage de charge : les *jobs* sont distribués de manière équitable aux calculateurs.

20.3.4 Résultats

Nous allons présenter et analyser une exécution de notre implémentation. Les machines utilisées sont des machines de bureau, en période d'utilisation dont les caractéristiques sont les suivantes :

Nom	Processeur	Système
`jago`	bi-pentium III 450 MHz	RedHat 9/Linux 2.4.7-10 SMP
`madiun`	UltraSparc 1 166 Mhz	SunOS 5.6
`bekasi`	Athlon XP 1900+ 1600 MHz	RedHat 7.2/Linux
`puceron`	Pentium III 450 Mhz	RedHat 8.0/Linux 2.4.18
`malang`	UltraSparc 1 166 Mhz	SunOS 5.8
`timor`	Pentium M 1400 MHz	Gentoo/Linux 2.4.26

Tous les conteneurs sont lancés à l'aide de la commande suivante :

```
1 shell> java -server                          # Pour de meilleurs performances
2 >mandala.jacob.remote.rmi.ServerRunner       # Lance un serveur
3 >ActiveMap                                   # de type ActiveMap
4 >RMIActiveMap                                # associé au nom RMIActiveMap
```

ce qui a pour effet d'utiliser l'implémentation naïve RMI de JACOb évitant ainsi le recours à un serveur LDAP[19] pour l'ensemble des conteneurs.

La version 1.4 du JDK est utilisée pour toutes les machines. Le client est aussi lancé sur `jago`. Aussi, pour ne pas surcharger celle-ci qui doit supporter à la fois l'équilibrage de charge, et le calcul des sous-sommes, une priorité de 1 sera spécifiée pour les threads associées à son gestionnaire (pour les autres la priorité par défaut est de 5). Par ailleurs, `jago` étant une machine bi-processeurs, deux threads seront associées à son gestionnaire.

20.3.4.1 Déroulement

Nous avons au commencement de l'application la sortie suivante :

[19] Un serveur `rmiregistry` est automatiquement lancé si rien n'est spécifié.

```
shell> java pi.PiDistributedComputer
>10000                                          # Premières décimales
>500                                            # Sous-sommes
>rmi://jago/RMIActiveMap;threads=2;priority=1 # Linux/bi-pentium III 450 Mhz
>rmi://madiun/RMIActiveMap                      # Solaris/UltraSparc 1 166 Mhz
>rmi://bekasi/RMIActiveMap                      # Athlon XP 1900+ 1600 MHz
>rmi://puceron/RMIActiveMap                     # Linux/Pentium III 450 Mhz
>rmi://malang/RMIActiveMap                      # Solaris/UltraSparc 1 166 Mhz
>rmi://timor/RMIActiveMap                       # Linux/Pentium M 1400 MHz

00.196 main [Log] Parsing rmi://jago/RMIActiveMap;threads=2;priority=1
01.305 main [Log] rmi://jago/RMIActiveMap: 2 threads with priority set to 1
01.307 main [Log] Parsing rmi://madiun/RMIActiveMap
01.531 main [Log] rmi://madiun/RMIActiveMap: 1 threads with priority set to 5
01.531 main [Log] Parsing rmi://bekasi/RMIActiveMap
01.640 main [Log] rmi://bekasi/RMIActiveMap: 1 threads with priority set to 5
01.641 main [Log] Parsing rmi://puceron/RMIActiveMap
01.758 main [Log] rmi://puceron/RMIActiveMap: 1 threads with priority set to 5
01.759 main [Log] Parsing rmi://malang/RMIActiveMap
01.973 main [Log] rmi://malang/RMIActiveMap: 1 threads with priority set to 5
01.974 main [Log] Parsing rmi://timor/RMIActiveMap
02.038 main [Log] rmi://timor/RMIActiveMap: 1 threads with priority set to 5
02.222 main [Log] Computing Pi to the 10000 th digits...
03.753 rmi://jago/RMIActiveMap,1 [Log] job 1 sent.
03.754 rmi://jago/RMIActiveMap,0 [Log] job 0 sent.
04.143 rmi://bekasi/RMIActiveMap,0 [Log] job 2 sent.
04.682 rmi://puceron/RMIActiveMap,0 [Log] job 3 sent.
04.793 rmi://madiun/RMIActiveMap,0 [Log] job 4 sent.
05.317 main [Log] Waiting ...
05.372 rmi://timor/RMIActiveMap,0 [Log] job 5 sent.
06.055 rmi://malang/RMIActiveMap,0 [Log] job 6 sent.
...
```

Comme on peut le voir, le mécanisme d'insertions asynchrones présenté en §20.3.3.1 p246 permet le démarrage des calculs au plus tôt : lorsque la thread principale a terminé les insertions et qu'elle se place en attente du résultat (caractérisé par la mention `Waiting...`), cinq *jobs* ont déjà été envoyés.

La suite du programme consiste en l'assignation des sous-sommes de manière séquentielle :

```
...
18:50.461 rmi://malang/RMIActiveMap,0 [Log] job 457 computed.
18:50.468 rmi://malang/RMIActiveMap,0 [Log] job 481 sent.
18:51.409 rmi://puceron/RMIActiveMap,0 [Log] job 472 computed.
18:51.416 rmi://puceron/RMIActiveMap,0 [Log] job 482 sent.
```

```
18:52.408 rmi://bekasi/RMIActiveMap,0 [Log] job 479 computed.
18:52.422 rmi://bekasi/RMIActiveMap,0 [Log] job 483 sent.
18:53.488 rmi://madiun/RMIActiveMap,0 [Log] job 460 computed.
18:53.495 rmi://madiun/RMIActiveMap,0 [Log] job 484 sent.
18:56.603 rmi://timor/RMIActiveMap,0 [Log] job 480 computed.
18:56.611 rmi://bekasi/RMIActiveMap,0 [Log] job 483 computed.
18:56.625 rmi://timor/RMIActiveMap,0 [Log] job 485 sent.
18:56.666 rmi://bekasi/RMIActiveMap,0 [Log] job 486 sent.
19:08.132 rmi://timor/RMIActiveMap,0 [Log] job 485 computed.
19:08.132 rmi://bekasi/RMIActiveMap,0 [Log] job 486 computed.
19:08.139 rmi://jago/RMIActiveMap,0 [Log] job 477 computed.
19:08.148 rmi://timor/RMIActiveMap,0 [Log] job 487 sent.
19:08.156 rmi://jago/RMIActiveMap,0 [Log] job 488 sent.
19:08.159 rmi://bekasi/RMIActiveMap,0 [Log] job 489 sent.
19:10.630 rmi://jago/RMIActiveMap,1 [Log] job 478 computed.
19:10.638 rmi://jago/RMIActiveMap,1 [Log] job 490 sent.
19:13.617 rmi://bekasi/RMIActiveMap,0 [Log] job 489 computed.
19:13.622 rmi://puceron/RMIActiveMap,0 [Log] job 482 computed.
19:13.632 rmi://bekasi/RMIActiveMap,0 [Log] job 491 sent.
19:13.661 rmi://puceron/RMIActiveMap,0 [Log] job 492 sent.
19:14.080 rmi://timor/RMIActiveMap,0 [Log] job 487 computed.
19:14.095 rmi://timor/RMIActiveMap,0 [Log] job 493 sent.
19:17.603 rmi://bekasi/RMIActiveMap,0 [Log] job 491 computed.
19:17.615 rmi://bekasi/RMIActiveMap,0 [Log] job 494 sent.
19:30.939 rmi://bekasi/RMIActiveMap,0 [Log] job 494 computed.
19:30.939 rmi://timor/RMIActiveMap,0 [Log] job 493 computed.
19:30.947 rmi://bekasi/RMIActiveMap,0 [Log] job 495 sent.
19:30.948 rmi://jago/RMIActiveMap,0 [Log] job 488 computed.
19:30.974 rmi://jago/RMIActiveMap,0 [Log] job 496 sent.
19:30.984 rmi://timor/RMIActiveMap,0 [Log] job 497 sent.
19:33.702 rmi://jago/RMIActiveMap,1 [Log] job 490 computed.
19:33.716 rmi://jago/RMIActiveMap,1 [Log] job 498 sent.
19:34.927 rmi://bekasi/RMIActiveMap,0 [Log] job 495 computed.
19:34.948 rmi://bekasi/RMIActiveMap,0 [Log] job 499 sent.
```

Dans cet extrait, on voit clairement les machines qui réalisent le plus de *jobs* : `bekasi` apparaît 14 fois, `malang` et `madiun`, une seule.

Lorsque le dernier *job* est assigné (le n°499 à `bekasi`), l'algorithme d'équilibrage de charge présenté dans la section précédente entre en scène :

```
19:35.384 rmi://puceron/RMIActiveMap,0 [Log] job 492 computed.
19:35.391 rmi://puceron/RMIActiveMap,0 [Log] every jobs are assigned!
19:35.397 rmi://puceron/RMIActiveMap,0 [Log] average time is 00:22.953
19:35.401 rmi://puceron/RMIActiveMap,0 [Log] job 481 will end in 00:08.610
19:35.401 rmi://puceron/RMIActiveMap,0 [Log] job 484 will end in 00:11.841
19:35.402 rmi://puceron/RMIActiveMap,0 [Log] job 496 will end in 00:20.426
```

```
19:35.402 rmi://puceron/RMIActiveMap,0 [Log] job 497 will end in 00:03.730
19:35.403 rmi://puceron/RMIActiveMap,0 [Log] job 498 selected: minimum
                                              remaining time is 00:23.179
19:35.406 rmi://puceron/RMIActiveMap,0 [Log] job 498 sent
```

`puceron` termine la première (*job* 492), et cherche le premier *job* dans la liste `computingJobs` qu'elle est susceptible de terminer en moins de temps. C'est le *job* 498 qui est sélectionné et envoyé, il est déjà assigné à `jago`. C'est ensuite `timor` qui demande un *job* :

```
19:36.968 rmi://timor/RMIActiveMap,0 [Log] job 497 computed.
19:36.984 rmi://timor/RMIActiveMap,0 [Log] every jobs are assigned!
19:36.984 rmi://timor/RMIActiveMap,0 [Log] average time is 00:08.136
19:36.985 rmi://timor/RMIActiveMap,0 [Log] job 481 will end in 00:07.027
19:36.985 rmi://timor/RMIActiveMap,0 [Log] job 484 selected: minimum
                                           remaining time is 00:10.257
19:36.986 rmi://timor/RMIActiveMap,0 [Log] job 484 sent.
```

Il récupère le *job* 484 déjà assigné à `madiun`. La machine la plus performante, `bekasi` termine en troisième et demande un *job* à son tour :

```
19:39.116 rmi://bekasi/RMIActiveMap,0 [Log] job 499 computed.
19:39.123 rmi://bekasi/RMIActiveMap,0 [Log] every jobs are assigned!
19:39.124 rmi://bekasi/RMIActiveMap,0 [Log] average time is 00:07.120
19:39.124 rmi://bekasi/RMIActiveMap,0 [Log] job 481 will end in 00:04.887
19:39.124 rmi://bekasi/RMIActiveMap,0 [Log] job 496 selected: minimum
                                            remaining time is 00:16.703
19:39.130 rmi://bekasi/RMIActiveMap,0 [Log] job 496 sent.
```

Elle récupère le *job* 496 qui est déjà assigné à `jago`. Quand vient le tour de `malang`, une des machines les moins performantes, l'équilibreur de charge ne trouve pas de *job* dont elle pourrait accélérer le calcul. C'est donc l'algorithme de tolérance aux pannes qui prend le relais :

```
19:41.560 rmi://malang/RMIActiveMap,0 [Log] job 481 computed.
19:41.579 rmi://malang/RMIActiveMap,0 [Log] every jobs are assigned!
19:41.580 rmi://malang/RMIActiveMap,0 [Log] average time is 00:53.432
19:41.580 rmi://malang/RMIActiveMap,0 [Log] job 484 will end in 00:03.541
19:41.581 rmi://malang/RMIActiveMap,0 [Log] job 496 will end in 00:04.665
19:41.581 rmi://malang/RMIActiveMap,0 [Log] job 498 will end in 00:16.775
19:41.581 rmi://malang/RMIActiveMap,0 [Log] Can't do a better job. Starting
                                            the fault tolerance algorithm
19:41.582 rmi://malang/RMIActiveMap,0 [Log] job 484 selected
                                            (Fault tolerance algorithm)
19:41.585 rmi://malang/RMIActiveMap,0 [Log] job 484 sent.
```

A ce stade, la liste `computingJobs` apparaît ; il reste les *jobs* :

- **484** assigné à `madiun` et `timor` et maintenant `malang` ;
- **496** assigné à `jago` et `bekasi` ;
- **498** assigné à `jago` et `puceron`.

Rappelons que le gestionnaire de `jago` dispose de 2 threads (machine bi-processeurs) ce qui explique que 2 *jobs* lui soit assigné. C'est `timor` qui termine le *job* 484 en premier[20], un peu plus d'1 seconde après l'assignation précédente à malang. Il lui appartient donc d'avertir les autres gestionnaires :

```
19:42.985 rmi://timor/RMIActiveMap,0 [Log] job 484 computed.
19:43.023 rmi://timor/RMIActiveMap,0 [Log] Cancelling future assigned to
                                             rmi://madiun/RMIActiveMap of
                                             job number 484
19:43.112 rmi://timor/RMIActiveMap,0 [Log] Cancelling future assigned to
                                           rmi://malang/RMIActiveMap of
                                           job number 484
```

Les objets `FutureClient` récupérés lors des appels asynchrones sont utilisés pour l'annulation (*cf.* les mécanismes d'annulation de RAMI présentés en §15.1.3 p160). La réponse est immédiate, la thread associée au gestionnaire de `madiun` qui a envoyé le *job* se réveille :

```
19:43.114 rmi://madiun/RMIActiveMap,0 [Log] Interrupted!!
```

Les threads des gestionnaires continuent ainsi à demander des *jobs* jusqu'à ce que la liste `computingJobs` soit vide :

```
...
19:54.318 rmi://jago/RMIActiveMap,1 [Log] Interrupted!!
19:54.321 rmi://jago/RMIActiveMap,1 [Log] every jobs are assigned!
19:54.322 rmi://jago/RMIActiveMap,1 [Log] average time is 00:24.865
19:54.322 rmi://jago/RMIActiveMap,1 [Log] job 498 is already assigned
                                            to this manager!
19:54.322 rmi://jago/RMIActiveMap,1 [Log] Can't do a better job.
                                          Starting the fault tolerance algorithm
19:54.323 rmi://jago/RMIActiveMap,1 [Log] No job selected
                                          (Fault tolerance algorithm)
00:19:54.325 rmi://jago/RMIActiveMap,1 [Log] STATS
...
```

[20] Pour être tout à fait exact, c'est la thread du gestionnaire de `timor` qui entre la première dans le bloc synchronisé de la boucle principale.

c'est la deuxième thread du gestionnaire associé à `jago` qui se rend compte qu'il n'y a plus de *job* à réaliser (inutile d'assigner le même *job* à une thread du même gestionnaire), et qui termine son exécution en affichant quelques statistiques. Toutes les threads terminent de la même façon. La dernière réveille la thread principale (`main()`) qui affiche les 10 000 premières décimales de π et les statistiques par conteneurs :

```
19:57.073 rmi://puceron/RMIActiveMap,0 [Log] i'm the last one,
                                             notifying the main Thread
19:57.074 main [Log] Awakened!
19:58.728 main [Log] Pi = 3.14159.....................37567
19:58.748 main [Log] Time stats:
Initialization time: 00:03.278
Distributed computing time: 19:51.757
Summing local results time: 00:00.029
Total: 19:55.064

19:58.748 main [Log] Average time par manager
19:58.748 main [Log] rmi://jago/RMIActiveMap: 00:24.865 (94 jobs computed)
19:58.749 main [Log] rmi://madiun/RMIActiveMap: 00:53.747 (21 jobs computed)
19:58.749 main [Log] rmi://bekasi/RMIActiveMap: 00:07.113 (166 jobs computed)
19:58.749 main [Log] rmi://puceron/RMIActiveMap: 00:22.953 (51 jobs computed)
19:58.750 main [Log] rmi://malang/RMIActiveMap: 00:53.432 (22 jobs computed)
19:58.750 main [Log] rmi://timor/RMIActiveMap: 00:08.161 (146 jobs computed)
19:58.750 main [Log] Removing remote objects
```

Ces statistiques vont nous permettrent d'analyser notre algorithme.

20.3.4.2 Analyse

L'ensemble des graphiques présentés sur la figure Fig. 20.6 p258 nous permet de tirer un certain nombre de conclusions sur le déroulement de notre exécution.

Le graphique (a) montre la répartition des 500 sous-sommes du calcul des 10 000 premières décimales de π au sein des threads des gestionnaires. Remarquons tout d'abord que le total fait 508, et non 500 en raison des assignations multiples qui ont lieu à la fin. Par ailleurs, en observant le temps nécessaire à un calculateur distant pour calculer une sous-somme (graphique (b)), on comprend mieux pourquoi `bekasi` et `timor` ont réalisé à elles seules 62 % du calcul.

Pour analyser la pertinence de notre algorithme d'équilibrage de charge, nous avons cherché à mesurer le temps d'*inactivité*, τ d'une thread t d'un gestionnaire donné :

$$\tau_t = \Delta - \delta_t$$

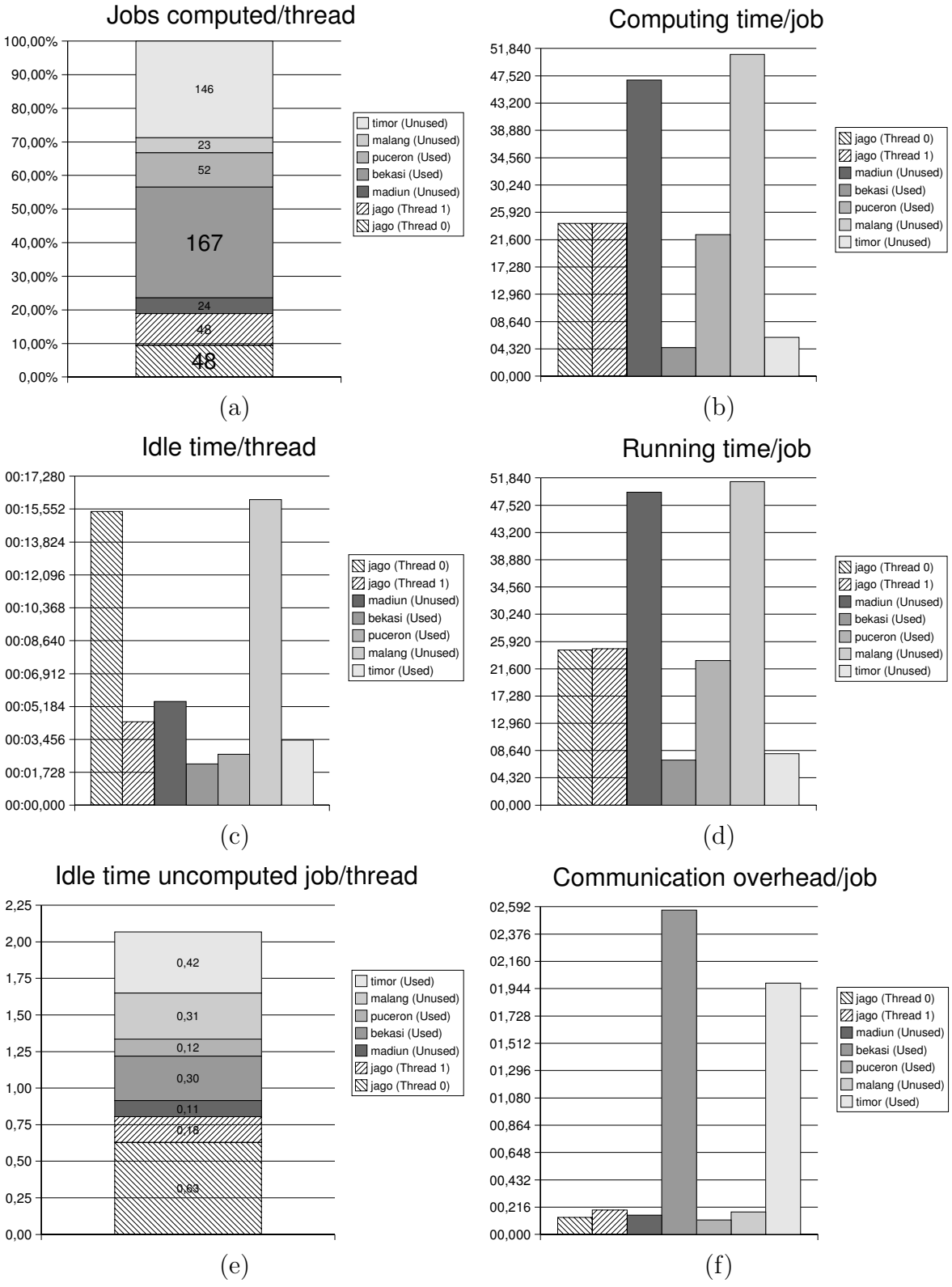


FIG. 20.6 – Analyse du calcul de π sur un réseau de machines hétérogènes *non-dédiées*.

où Δ est le temps total du calcul, et δ_t, la durée de vie de la thread t. Remarquons que τ est une majoration, puisque le temps total comprend le temps d'initialisation (insertion dans les conteneurs), le temps distribué (calcul parallèle), et le temps d'accumulation (la somme finale des sous-calculs qui donne le nombre π). Le graphique (c) représente ces résultats et montre que chaque thread a un temps d'inactivité qui lui est propre, indépendamment de la performance de la machine associée à son gestionnaire : les deux threads associées à `jago` par exemple, ont des temps d'inactivité très différents. La compétition des threads pour l'acquisition des verrous, les interventions sporadiques du ramasse-miettes, les aléas liés à l'utilisation des machines et aux interventions du système d'exploitation sous-jacent sont des facteurs qui rendent l'ordonnancement des threads relativement indéterministe.

Connaissant τ pour chaque thread, nous pouvons estimer le nombre de *jobs* qui *auraient pu être calculés*. En effet, on peut considérer que si une thread t est restée inactive pendant une durée cumulée de τ_t, elle aurait peut-être pu, avec un bon algorithme calculer quelques *jobs* supplémentaires. Il convient donc de mesurer le temps nécessaire à la thread t pour calculer 1 *job* (temps d'acquisition de verrous, de sélection de *job*, d'envoi et de retour du job, et temps de calcul des statistiques inclus). Pour cela, le temps de calcul total pour 1 job, ρ_t est donné par

$$\rho_t = \delta_t / N_t$$

où N_t est le nombre de *jobs* calculés par la thread t. Le graphique (d) présente ces données.

Il est donc maintenant possible de mesurer la pertinence de notre algorithme. Le graphique (e) présente le nombre de *jobs* que chaque thread aurait pu calculer durant sa péridode d'inactivité. Une quantité supérieure à 1 n'est jamais obtenue, ce qui montre que notre algorithme est très pertinent. Par ailleurs, le total cumulé étant légèrement supérieur à 2, il est assez loin du maximum acceptable qui est proche de 7 (si chaque thread avait presque pu faire 1 *job*).

Nous pouvons enfin évaluer le *gain* obtenu par la distribution en comparant l'exécution parallèle et une exécution séquentielle. Dans un cadre homogène, le gain est défini par le *speedup* S :

$$S = \frac{BestSequentialTime}{BestParallelTime}$$

Nous connaissons le temps parallèle, mais le problème est de définir "le meilleur temps séquentiel" dans un cadre hétérogène. Nous proposons trois mesures : le temps du calcul des 10 000 premières décimales de π sur la meilleure machine (`bekasi`), en utilisant l'option `-client` et `-server`, ainsi qu'une estimation du temps de calcul des 500 *jobs* nécessaires en partant du temps de calcul d'1 job. Enfin, l'efficacité

Machine	Poids
`jago`	0.3757
`madiun`	0.0969
`bekasi`	1.0000
`puceron`	0.2029
`malang`	0.0892
`timor`	0.7414
Total	2.5060

TAB. 20.1 – Poids relatif des différentes machines utilisées pour le calcul des décimales de π.

	`-client`	**`-server`**	Estimation
Temps	01:00:02.619	**00:33:27.898**	00:37:59.802
Gain (temps)	00:40:07.555	**00:13:32.834**	00:17:54.738
Speedup	3.01	**1.68**	1.9
Efficacité	1.2	**0.67**	0.76

TAB. 20.2 – Gain obtenu par l'utilisation d'un réseau de machines hétérogènes *non-dédiées* pour le calcul des 10 000 premières décimales de π.

dans un cadre hétérogène peut-être défini par [225] :

$$E = \frac{S}{\sum_{j=1}^{p} W_j}$$

où W_j est le *poids* de la machine j dans le réseau. Ce *poids* est défini par :

$$W_i = \frac{min\{T_j\}}{T_i}$$

où T_j est le temps d'exécution d'un *job* par la machine j (à ne pas confondre avec ρ_t qui est le temps d'exécution total d'un *job* par la thread t d'un gestionnaire.). Le poids des machines utilisées[21] est présenté dans la table TAB. 20.1 p260 et les métriques dans le tableau TAB. 20.2 p260.

Une efficacité de 0.67 est assez décevante. Elle met en lumière le problème lié à l'utilisation de machine *non-dédiée*. En particulier, `timor` est un portable qui se trouve être sur un sous-réseau du réseau principal pour des raisons de sécurité. De même, des problèmes de connexions réseau semblent avoir pénalisé `bekasi` : cela se traduit sur le graphique (f) de la figure FIG. 20.6 p258 qui représente la latence

[21] Le poids de la machine `jago` a été multiplié par deux puisqu'elle dispose de deux processeurs.

moyenne par *job*, c'est à dire, la différence entre le temps de l'invocation distante de la méthode `compute()` et le temps réel de son exécution. Clairement, les deux machines les plus performantes ont des latences particulièrement importantes[22]. Cette latence n'est pas prise en compte dans le calcul de l'efficacité ce qui explique ce résultat relativement bas. Une alternative pour recouvrir les temps de communication par du calcul est d'anticiper la fin de l'appel à la méthode `compute()` et d'envoyer un *job* juste avant. Ainsi, lorsqu'un *job* est terminé, un calculateur n'a pas à attendre l'envoi d'un nouveau *job* pour recommencer à faire quelque chose.

20.4 Bilan

De nombreuses autres applications ont été développées à l'aide de Mandala (*branch and bound* pour la résolution du problème NP-Complet du voyageur de commerce [153] ; application d'analyse spectrale distribuée de son ; plate-forme de type *bag of tasks* [213] ; calcul fractal parallèle et/ou distribué [211], ...) mais nous nous limiterons au trois exemples présentés dans ce chapitre. Ils couvrent en effet une grande partie de l'éventail des possibilités de notre plate-forme :

- **l'exemple des agents mobiles** met en évidence le concept des conteneurs actifs qui est sous-jacent ;
- **l'explorateur de classes** souligne l'importance de l'aspect *dynamique* de Mandala en utilisant une bibliothèque dans un cadre asynchrone ;
- **le calcul parallèle des décimales de π** témoigne de la relative facilité du développement d'applications distribuées.

De nombreuses extensions sont prévues, pour faciliter le développement. Par exemple, la possibilité de préciser une priorité dans un appel asynchrone devrait être apportée prochainement à RAMI. En ce qui concerne JACOb, un mécanisme assurant la sécurité (*cf.* §10.6.2 p115), devrait être proposé en utilisant la clé d'un *stored object* pour stocker des informations (mot de passe par exemple).

[22]Ce qui est aussi clairement mentionné dans les *logs* : une différence de 14 secondes (!) a été mesuré entre la meilleure et la moins bonne latence pour les *jobs* soumis à `bekasi`. Pour `timor`, la différence est de 13 secondes.

Chapitre 21

Conclusion

Nous avons présenté la plate-forme Mandala [212] globalement. De nombreuses techniques ont été mise en œuvre pour son développement. Elle offre, conceptuellement, une alternative intéressante aux solutions actuelles pour le développement d'applications concurrentes et/ou distribuées orientée objets. Cependant, il reste des problèmes majeurs pour son utilisation et son déploiement à grande échelle, en particulier en terme de sécurité et de performance.

Nous avons présenté quelques applications qui utilisent notre plate-forme afin d'illustrer la mise en œuvre des mécanismes que nous avons vu tout au long de ce document (appel distant, appel asynchrone, réponse anticipée (*future*), récupération du résultat par évènement (*callback*), annulation, transparence, réflexion, insertions asynchrones, déploiement, ...). Surtout, nous espérons que l'aspect *dynamique* a été clairement mis en évidence par ces exemples : les objets que nous avons manipulé de manière asynchrone et/ou distante n'ont pas été prévu à cet effet.

Mandala est distribuée avec la GNU LESSER GENERAL PUBLIC LICENSE (LGPL) Version 2.1 de février 1999 [69] de la Free Software Foundation (FSF). Elle est documentée (*Javadoc* et *PDF User's Guide*), et hébergée sur SourceForge : `http://mandala.sf.net`. Au 12 juillet 2004, le site a été visité 5 798 fois et 183 téléchargements ont été effectués. C'est assez peu. Communiquer reste un des enjeux majeur dans le développement *open-source*. Et nous devons admettre que de gros efforts sont à fournir dans ce domaine pour que la plate-forme soit mieux reconnue.

Sixième partie

Conclusion générale et perspectives

Chapitre 22

Conclusion

22.1 Mobilité

Au début de cette thèse, le domaine du code mobile et plus particulièrement celui des agents mobiles était en pleine effervescence ; une révolution dans le domaine informatique semblait imminente ; des programmes autonomes, relativement intelligents, devaient foisonner sur Internet et migrer de machine en machine afin de réaliser un travail pour le compte d'un individu ou d'un autre programme. Force est de constater qu'à part les virus et autres vers, ce n'est pas le cas : les principales plates-formes présentées en §4.3 p44 ne sont d'ailleurs plus maintenues. A cela nous avons énuméré plusieurs raisons d'ordre technique (sécurité, interopérabilité) ou non (mode de rémunération, qualité de service, performance, maturité) (*cf.* §4.2 p42).

Finalement, Java ne semble pas être le langage idéal pour l'implémentation d'une plate-forme d'agents mobiles [179]. En particulier, Java possède les problèmes suivants :

- absence de contrôle des ressources (CPU, mémoire, réseaux) ; une extension du langage n'est pas prévu à cet effet dans un avenir proche ; des travaux dans ce domaine sont menés par l'équipe *Composants* du laboratoire *Valoria* dans le projet SAJE [62] ;
- absence de séparation des applications ; les *isolats* (JSR 121 [11]) semblent être une solution, mais elle ne sera pas disponible avant le JDK v1.6 ;
- absence de mécanisme permettant *l'interruption autoritaire de thread* (*cf.* §5.2.3.1 p58 et §C.3.4.2 p309) ;
- le ramasse-miettes peut être "*piraté*" en surchargeant la méthode `finalize()` ;
- le fonctionnement de certaines threads internes à la machine virtuelle peut être bloqué aprés l'acquisition de verrous de classes déclarées `public`.

22.2 Résultats

Notre étude du code mobile, nous a amené à une modélisation d'un système d'agents mobiles qui a révélé une nouvelle structure de données : le *conteneur actif*. Nous avons vu qu'elle fournit une entité de base à partir de laquelle une application distribuée peut être écrite. Nous avons implémenté ce concept en Java. L'utilisation de cette implémentation nous a dirigé vers l'expression de la concurrence dans une application orientée objets et vers un nouveau concept : la *référence asynchrone*. Nous avons aussi étudié les mécanismes de transparence dans le paradigme d'appel de méthode asynchrone ; nous sommes arrivé à deux conclusions :

- la transparence totale des appels de méthode asynchrone est à proscrire (*cf.* §16.1.4 p185) ;
- la semi-transparence offre une solution raisonnable (*cf.* §16.3 p193).

L'ensemble de nos travaux se présente sous la plate-forme appelée Mandala qui est documentée, disponible en licence libre LGPL sur `http://mandala.sf.net`.

Mandala facilite l'implémentation des applications distribuées orientées objets, comme nous l'avons montré sur de nombreux exemples.

22.3 Perspectives

En premier lieu, une preuve manque à nos travaux : le système d'agents mobiles modélisé au dessus du modèle des conteneurs actifs en §D.1 p315 est-il équivalent[1] à celui modélisé en C.3.3 ? Cette preuve renforcera les fondements de notre concept et de la plate-forme. Il est par ailleurs souhaitable d'implémenter les concepts de conteneurs actifs et de références asynchrones dans un autre langage. Nous avons vu en effet un certain nombre de limitations qui sont directement liées au langage Java et à la machine virtuelle sous-jacente.

En ce qui concerne Mandala, il y a plusieurs *Request Feature Enhancement (RFE)* sur le site [215] qui demande à être développé (génération des proxys semi-transparents à partir du code source, JACOb/Myrinet, JACOb/HTTP, entre autres). Par ailleurs, un certain nombre de services (*cf.* §10.6 p113) manquent dans la plate-forme. En particulier un service de sécurité empêche la plate-forme d'être déployée à grande échelle. Une analyse du code doit aussi être réalisée avec des outils adaptés (*profiler*) afin d'améliorer les performances d'un certain nombre d'implémentations (*cf.* §19 p211).

Les applications envisagées sont nombreuses. La première, qui est la suite logique de ces travaux (ou qui aurait pu être mené en parallèle), est un outil d'analyse de

[1] Le sens de "équivalent" devra être clairement défini par une bisimulation appropriée (*cf.* définition B.3.12 page 289).

code qui transforme toute référence Java en référence asynchrone. Cet outil doit ensuite être capable de réordonner les instructions en utilisant la règle §16.1.2 p185. Enfin, un autre outil doit être capable de déterminer selon un critère ressemblant au critère d'activabilité des objets actifs (*cf.* §12.4.1.1 p137) si une référence peut être une référence sur un *stored object distant* ou non. De cette manière l'application doit pouvoir garder sa sémantique en étant exécutée de manière parallèle.

Évidemment, d'autres applications peuvent tirer parti de Mandala. Citons entre autres le déploiement d'applications, le *peer to peer* ou le calcul opportuniste. Nous avons vu le potentiel des conteneurs actifs en §9.11 p90. Couplés aux références asynchrones, nous pensons que le modèle de programmation que nous proposons est à même de simplifier le développement de n'importe quel type d'application concurrente et/ou distribuée.

22.4 Bilan général

De manière très générale, deux grandes conclusions peuvent être tirées de nos travaux :

dynamisme : les aspects distants et asynchrones doivent être orthogonaux aux aspects métiers pour apporter du *dynamisme* et faciliter la réutilisation des classes dans des contextes différents. C'est en particulier le cheval de bataille de la programmation orientée aspects [110] que de rendre orthogonaux tous les aspects qui ne sont pas métiers. Notre proposition à base de conteneurs actifs et de références asynchones a l'avantage d'être simple et donc facilement assimilée.

transparence : la transparence est une propriété qui doit faciliter le développement des applications sans en modifier la pratique. Nous avons vu en §10.2 p98 que l'aspect distant ne pouvait pas être transparent sans restrictions importante du cadre d'utilisation. La notion de conteneurs actifs permet de prendre en compte l'aspect distant de manière non-transparente, globale, et indépendante des objets qu'il contient. En ce qui concerne les appels asynchrones, la partie §16 p179 a présenté les problèmes d'une démarche qui tente de les rendre totalement transparent. Notre proposition, la semi-transparence, est un compromis qui rend l'écriture des appels asynchrones aussi facile que les appels synchrones tout en assurant la non-transparence de l'asynchronisme (*cf.* Conscience du développeur en §16.1.4.2 p185).

Annexes

Annexe A

Formalismes

Afin de modéliser le paradigme *agents mobiles* présenté en §4 p41, nous devons choisir un outil pour la modélisation. Cette annexe propose une courte description des différents formalismes étudiés et les raisons qui nous ont orienté vers le π-calcul pour notre modèle. On trouvera en [157] un site dédié au calcul de processus mobiles.

A.1 Le π-calcul

Ce modèle de calcul proposé par Milner, Parrow et Walker [178] vise à améliorer le pouvoir d'expression de CCS [149] en autorisant le passage de "canaux" entre processus. Une première extension de CCS nommée ECCS avait été présentée par Engberg et Nielsen [63] pour permettre le passage d'étiquettes (*labels*). Cependant, le π-calcul simplifie et généralise ses prédécesseurs. Depuis, beaucoup de travaux ont porté sur le π-calcul : ils visent soit à améliorer son pouvoir d'expression, soit à mieux comprendre la théorie sous-jacente. Les travaux de D. Sangiorgi [180] en particulier nous ont permis de mieux comprendre le π-calcul. Ils sont décrits ci-dessous.

A.1.1 L'étude de D. Sangiorgi

Cette étude permet de mieux assimiler les difficultés du π-calcul (principalement la dissymétrie entre l'envoi et la réception d'un nom). Sangiorgi définit un calcul – noté πI – basé sur le π-calcul mais limité à la *mobilité interne i.e.* dont l'envoi de noms *libre* est interdit (*cf.* §B.2.2 p280). Cette mobilité est responsable de la majorité du pouvoir d'expression du π-calcul tandis que la *mobilité externe i.e.* l'envoi de noms liés est responsable de la majorité de sa complexité sémantique. Par ailleurs, une propriété agréable de πI est la symétrie complète entre l'envoi et la réception de noms contrairement au π-calcul. En outre, Sangiorgi montre que l'ajout de l'ordre supérieur dans le π-calcul n'augmente pas son pouvoir d'expression.

A.1.2 Polymorphisme

Pierce et Sangiorgi dans [167] proposent une extension du π-calcul qui gère le polymorphisme. L'idée est d'autoriser l'envoi supplémentaire d'un type avec l'envoi d'un nom. Ainsi, il est possible d'écrire des expressions qui reçoivent une valeur sans en connaître le type.

A.1.3 Raffinement du typage

Plusieurs études portent sur le typage du π-calcul. Par exemple, Pierce et Sangiorgi étendent dans [166] la syntaxe du π-calcul en définissant un sous-typage. Un canal peut être de type "lecture seulement", "écriture seulement" ou les deux, limitant un canal respectivement à recevoir des données, envoyer des données ou les deux. Le polymorphisme est introduit dans cet étude mais n'est pas développé autant que dans la précédente qui est plus récente.

Une autre étude concernant le typage du π-calcul est présentée par Pierce, Kobayashi et Turner dans [114]. Cette étude étend le sous-typage précédent qui contient la notion de polarité (écriture seule, lecture seule, ou lecture-écriture) en définissant un critère de multiplicité pour les canaux de communications. Un canal pourra être utilisé un nombre de fois borné dans le sens correspondant à sa polarité. Cela permet d'exprimer des limites d'accés à des ressources.

A.1.4 Asynchronisme

Les travaux de Boudol [35] ont pour but de définir le π-calcul à partir d'un modèle de calcul de plus bas niveau (une machine chimique abstraite *cf.* §A.4 p275) dans lequel le passage de messages s'effectue de manière asynchrone. Le passage de messages synchrones pouvant être simulé par le passage de messages asynchrones à l'aide d'un signal d'accusé de réception, le π-calcul peut être codé dans ce nouveau modèle de calcul. Cette étude diffère de celle effectuée par Honda et Tokoro [100] dans la mesure où elle n'utilise pas de notion de bisimulation mais une adaptation de l'observabilité des λ-termes.

A.1.5 Outils

Le π-calcul est l'objet de nombreuses recherches et plusieurs outils sont en cours de développement pour faciliter sa compréhension et l'étude de modèles écrits dans ce formalisme. Citons par exemple les travaux de Melham [134] qui donnent une méthode pour axiomatiser le π-calcul dans l'assistant de démonstration HOL [200].

Le π-calcul peut être comparé au λ-calcul inventé par Church [52]. L'influence de ce dernier en informatique est très importante puisqu'il est à l'origine de la famille des langages de type LISP. De la même manière, CCS et le π-calcul sont à l'origine de familles de langage. CCS est associé au langage Facile [206], tandis que le π-calcul est implémenté dans le langage Pict [168]. Ce dernier propose certaines extensions déjà mentionnées (polymorphisme et sous-typage).

Nous n'avons pas trouvé autant de documentation sur un autre langage, autant de recherches et d'outils. Par ailleurs, le π-calcul est suffisamment expressif pour modéliser les agents mobiles en Java. Il est de plus assez intuitif de modéliser du code Java dans ce formalisme comme nous le verrons en §C.3.3 p299.

A.2 CHOCS

CHOCS [205] (Calculi for Higher Order Communicating Systems) est un modèle de calcul présenté par Thomsen. Ce modèle est strictement d'ordre supérieur puisqu'il n'accepte que le passage de termes entre processus. Ce modèle aurait pu être utilisé mais il strictement moins expressif que le π-calcul (*cf.* [180] section 7.3 p26).

A.3 Le spi-calcul

Le spi-calcul [64] étend le π-calcul avec des primitives de cryptographie. Nous n'avons pas vu d'intérêt à l'utiliser pour notre travail.

A.4 La Machine chimique abstraite

La machine chimique abstraite [30] n'est pas un formalisme à proprement parler. Ce modèle propose une syntaxe et une sémantique très différentes de CCS et de ses dérivés. Il utilise la notion de molécules et de réactions chimiques afin de donner une structure sémantique utilisable pour définir (ou coder) de nouveaux calculs.

A.5 Le join-calcul

Le join-calcul [74] est une reformulation du π-calcul avec une machine chimique abstraite. La notion de place d'interaction est mieux définie. Ce formalisme a donné naissance au langage du même nom. Des dérivés de ce calcul existent et peuvent être intéressants pour notre modélisation (notamment le join-calcul distribué [75]). De plus, le join-calcul est strictement équivalent au π-calcul. L'un est codable en l'autre

et vice-versa. Malheureusement, nous n'avons pas pu étudier une modélisation des agents mobiles en Java dans ce formalisme.

A.6 Les Ambiances mobiles

Le calcul basé sur la notion d'ambiances mobiles [40] est intéressante. Il fournit une puissance d'expression suffisante pour coder le π-calcul. Il propose la notion d'intérieur et d'extérieur d'une ambiance dans laquelle s'exécutent plusieurs processus qui peuvent y entrer ou en sortir. Surtout, Il sépare les primitives de communication des primitives de mobilité. Aussi, il paraît *a priori* plus simple de modéliser un système d'agents avec ce formalisme. Cependant, notre système d'agents mobiles était déjà modélisé quand nous l'avons découvert.

A.7 Conclusion : choix du π-calcul

Le π-calcul a été choisi principalement pour les raisons suivantes :

- il permet une modélisation assez proche de notre implémentation ;
- il reprend les principes syntaxiques de CCS dont il est un dérivé facilitant son apprentissage[1] ;
- il est l'objet de nombreux travaux pour le rendre plus expressif, nous en avons présenté quelques-uns ;
- une abondante littérature est disponible ce qui est un atout non négligeable ;
- plusieurs outils sont développés pour étudier les modèles écrits en π-calcul ou pour créer un langage de programmation inspiré du π-calcul.

[1] CCS est utilisé depuis longtemps.

Annexe B

Introduction au π-calcul

Le π-calcul est un modèle de calcul de processus concurrents basé sur la notion de *nommage*. Il a été présenté dans sa version d'origine sous sa forme monadique dans [178]. Puis, Robin Milner l'a étendu en une version polyadique dans [150] justifiée par la notion de *sorte* (typage) qui résoud les problèmes liés à l'encodage de la version polyadique en monadique. Dans cette nouvelle version, il simplifie la sémantique en définissant séparément les règles de réduction et le système de transitions étiquetées appelé *engagement*. C'est cette deuxième version du π-calcul, plus complète et plus précise que la première, que nous reprenons ici.

B.1 Introduction

Le π-calcul est le fruit d'un travail visant à améliorer le pouvoir d'expression de CCS[1]. Il est l'extension logique des travaux de Engberg et Nielsen [63]. C'est pourquoi il existe une grande ressemblance syntaxique entre ces deux modèles de calculs. Toutefois, le π-calcul est d'une complexité mathématique bien plus grande que CCS. Il apporte essentiellement deux nouveautés :

- la suppression des notions de variables, de valeurs et de noms de canaux au profit de la seule notion de nom ;
- le passage de noms entre processus, CCS se restreignant au passage de valeurs.

Ces deux nouveautés sont à l'origine de la puissance d'expression du π-calcul.

[1] Le lecteur n'ayant aucune connaissance de CCS pourra se référer à [149].

B.2 Le π-calcul monadique

B.2.1 Syntaxe

Comme nous l'avons vu, la plus petite entité primitive dans le π-calcul est le nom. Une autre entité est le processus.

Définition B.2.1 (Entités du π-calcul monadique)
Soit $\mathcal{X} = \{x, y, \ldots\}$ *l'ensemble infini des noms. Les noms ne sont pas structurés. Soit* $\mathcal{Q} = \{P, Q, \ldots\}$ *l'ensemble des processus.*

Si I est un ensemble fini d'index, les processus sont construits selon la syntaxe suivante :

Règles syntaxiques B.2.1 (π-calcul monadique d'ordre inférieur simple)

$$P ::= \sum_{i \in I} \pi_i.P_i \Big| P|Q \Big| !P \Big| (\nu\ x)P$$

Notation B.2.1 (Processus inactif)
Si $I = \emptyset$*, la somme* $\sum_{i \in I} \pi_i.P_i$ *est notée* $\mathbf{0}$.

Dans l'expression $\pi.P$, π est une *action atomique* : c'est la première action effectuée par P. Il existe deux formes de préfixes :

Définition B.2.2 (Sujets et objets d'une action)
Si $\pi = x(y)$*,* x *est un* sujet positif.
Si $\pi = \overline{x}y$*,* $\overline{x}$ *est un* sujet négatif. *Dans les deux cas,* y *est l'*objet *de l'action. On appellera aussi* $\overline{x}$ *le* co-nom *de* x *et l'on a* $\overline{\overline{x}} = x$*. Deux sujets sont* complémentaires *si l'un est le co-nom de l'autre. Deux processus sont* complémentaires *s'ils sont préfixés par deux sujets complémentaires.*

Le sens de ces préfixes est intuitivement le suivant :

- $x(y)$ lie y dans le processus préfixé et signifie : "attend une entrée – appelons là y – sur le canal de nom x" ;
- $\overline{x}y$ ne lie pas y dans le processus préfixé et signifie "envoie le nom y sur le canal de nom x".

La somme $\sum_{i \in I} \pi_i.P_i$ représente un processus capable de prendre part à une – et seulement une – parmi plusieurs possibilités de communications. Le processus ne peut choisir ; il ne peut s'engager dans une alternative avant qu'elle n'ait lieu, et celle-ci empêche tout engagement dans une autre. Les processus sous cette forme sont appelés *processus normaux*. En effet, Milner démontre dans [150] que tous processus peut être converti sous cette *forme normale*.

Définition B.2.3 (Processus normaux)
L'ensemble des processus normaux est $\mathcal{N} = \{M, N, \ldots\}$.

Leur syntaxe est la suivante:

Règles syntaxiques B.2.2 (Processus normaux)

$$N ::= \pi.P \Big| \mathbf{0} \Big| M + N$$

L'expression $P|Q$ – "P parallèle Q" – symbolise une activité concurrente des processus P et Q. Ils peuvent agir indépendemment l'un de l'autre (par des communications internes) mais peuvent aussi communiquer.

Définition B.2.4 (Réplication)
L'expression $!P$ *– "réplique* P*" – est définie ainsi:*

$$!P \stackrel{\text{def}}{=} P|!P$$

L'opérateur "!" est appelé *réplicateur*. Une de ses utilisations courantes est l'expression $!\pi.P$ qui permet de modéliser que l'accès à une ressource illimitée nécessite une communication via π.

Définition B.2.5 (Restriction)
L'opérateur ν *permet de restreindre un nom à un processus :*

$$(\nu\ x)P$$

x *n'est alors connu que du processus* P.

L'opérateur ν ne correspond pas tout à fait à l'opérateur de restriction de CCS. En effet, cet opérateur crée un nouveau nom. Et de par cette création, le nom n'est connu d'aucun autre processus. Toutefois, rien n'empêche un processus d'envoyer sur un nom global, un nom nouvellement créé via l'opérateur ν. Par exemple, si l'on a :

$$P \equiv (\nu\ y)\ \overline{x}y.P' \text{ et } Q \equiv x(z).Q'$$

alors dans l'expression $P|Q$, Q recevra y via x et Q' connaîtra y qui est pourtant restreint à P. C'est ce que l'on appelle *expulsion de portée* (scope extrusion en anglais).

Définition B.2.6 (Processus inactif)
Un processus est inactif *s'il ne peut communiquer avec aucun autre processus.*

Exemple B.2.1 (Processus inactif)
Les processus $(\nu\ x)(\nu\ z)\overline{x}z$ et $(\nu\ x)x.P$ sont inactifs. En effet, pour le premier, le nom x n'est connu d'aucun autre processus. Ils ne pourront donc pas recevoir le nom z via x. De même, le second ne peut recevoir sur un canal qu'il a lui-même créé, en l'occurrence ici x.

Les processus comme $x(y).\mathbf{0}$ et $\overline{x}y.\mathbf{0}$ sont si courants que nous omettrons le $.\mathbf{0}$ final et écrirons simplement $x(y)$ et $\overline{x}y$. De façon générale :

Notation B.2.2

$$P.\mathbf{0} = P$$

B.2.2 Congruence structurelle

Comme nous l'avons vu, il existe deux opérateurs liants : le sujet préfixe positif $x(y)$ et la restriction $(\nu\ x)$. Nous pouvons définir l'ensemble des *noms libres* noté $fn(P)$ (*free name* en anglais), l'ensemble des *noms liés* noté $bn(P)$ (*bound names*) et l'ensemble des noms d'un processus P noté $n(P)$:

Définition B.2.7 (Noms libres, noms liés et ensemble de noms)

$$\begin{aligned} bn(x(y)) = \{y\} \quad &, \quad fn(x(y)) = \{x\} \\ bn(\overline{x}y) = \emptyset \quad &, \quad fn(\overline{x}y) = \{x, y\} \\ n(P) \quad &\stackrel{\text{def}}{=} \quad bn(P) \cup fn(P) \end{aligned}$$

Rappelons quelques définitions :

Définition B.2.8 (Monoide)
Soit E un ensemble, $*$ une opération associative et e un élément neutre pour $*$. Alors $(E, *, e)$ est un monoide. *Si la loi $*$ est commutative, le monoide est dit commutatif.*

Robin Milner définit alors la *congruence structurelle* – noté $\equiv$ – ainsi :

Définition B.2.9 (Congruence structurelle $\equiv$)
La congruence structurelle $\equiv$ est la plus petite relation sur $\mathcal{P}$ vérifiant :

1. *des processus sont identiques s'ils ne diffèrent que par un changement de noms liés ;*
2. *$(\mathcal{N}/\equiv, +, \mathbf{0})$ est un monoide commutatif ;*
3. *$(\mathcal{P}/\equiv, |, \mathbf{0})$ est un monoide commutatif ;*
4. *$!P \equiv P|!P$;*

5. $(\nu\ x)\mathbf{0} \equiv \mathbf{0}, (\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$;
6. *Si* $x \notin fn(P)$ *alors* $(\nu\ x)(P|Q) \equiv P|(\nu\ x)Q$.

La congruence structurelle implique en particulier les propriétés suivantes :

1. Si $x \notin fn(P),\ (\nu\ x)P \equiv P$;
2. Toute restriction peut être factorisée dans un processus qui n'est pas en forme normale.

Afin d'habituer le lecteur au π-calcul, nous allons donner la preuve de la première propriété :

Preuve B.2.1 (Propriété 1)

$$\begin{aligned}(\nu\ x)P &\overset{(3)}{\equiv} (\nu\ x)(P|\mathbf{0}) \\ &\overset{(6)}{\equiv} P|(\nu\ x)\mathbf{0} \\ &\overset{(5)}{\equiv} P|\mathbf{0} \\ &\overset{(3)}{\equiv} P\end{aligned}$$

Pour la deuxième propriété, nous donnerons simplement un exemple :

Exemple B.2.2 (Propriété 2 : factorisation des restrictions)
Si $P \equiv (\nu\ y)\overline{x}y$, alors

$$\begin{aligned}x(z).\overline{y}z|!P &\overset{(4)}{\equiv} x(z).\overline{y}z|(\nu\ y)\overline{x}y|!P \\ &\overset{(1)}{\equiv} x(z).\overline{y}z|(\nu\ y')\overline{x}y'|!P \\ &\overset{(6)}{\equiv} (\nu\ y')(x(z).\overline{y}z|\overline{x}y')|!P\end{aligned}$$

qui après communication donne

$$(\nu\ y')(\overline{y}y'|\mathbf{0})|!P \equiv (\nu\ z)(\overline{y}z)|!P$$

Des écritures comme

$$A(x) \overset{\text{def}}{=} x(y).B(y)\ ; \qquad B(y) \overset{\text{def}}{=} \overline{y}z.A(z)$$

sont possibles. Milner montre en effet qu'il n'est pas nécessaire de rajouter des constructions syntaxiques au π-calcul : il présente un codage de la récursion en réplication que nous ne présenterons pas ici.

B.2.3 Règles de réduction

Avant d'aller plus loin, définissons la notion de substitution :

Définition B.2.10 (Substitution)
On note $\sigma = z/y$ une substitution, c'est à dire le remplacement syntaxique de y par z. $P\sigma$ est l'application de cette substitution à P.

Nous allons définir la *relation de réduction* $\rightarrow$ sur l'ensemble des processus. $P \rightarrow P'$ signifie que P peut être transformé en P' en une seule phase de réduction. Chaque phase est le résultat d'une interaction entre deux termes en forme normale. La première règle de réduction est la *communication* :

$$\text{COMM} : (\cdots + x(y).P)|(\cdots + \overline{x}z.Q) \rightarrow P\{z/y\}|Q$$

La communication survient donc entre deux processus normaux atomiques $\pi.P$ dont les sujets sont complémentaires. De plus, cette communication annihile toute autre possibilité.

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu\ x)P \rightarrow (\nu\ x)P'}$$

indiquent que la communication peut survenir *sous* la composition et la restriction. Finalement,

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

précise que deux processus structurellement congruents ont la même réduction.

L'ensemble de ces quatre règles se trouve ci-dessous :

$$\text{COMM} : (\cdots + x(y).P)|(\cdots + \overline{x}z.Q) \rightarrow P\{z/y\}|Q$$

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu\ x)P \rightarrow (\nu\ x)P'}$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Remarquons ce que ces règles *ne permettent pas*. Tout d'abord, la réduction ne peut être effectuée *sous* un préfixe ou une somme. Par conséquent, les préfixes imposent un ordre de réduction. Ensuite, les règles ne permettent pas la réduction *sous* une réplication.

Exemple B.2.3 (Réductions non permises)

$$u(v).(x(y)|\overline{x}z) \not\rightarrow$$

car le processus $(x(y)|\overline{x}z)$ est préfixé (nous emploierons le terme *gardé* en §B.2.11 p283). Il doit recevoir sur le canal u avant de continuer.

$$\text{Si } P \rightarrow P' \text{ alors } !P \not\rightarrow !P' \text{ mais}$$
$$!P \equiv \underbrace{P|P|\ldots|P}_{n}|!P \xrightarrow{n} \underbrace{P'|P'|\ldots|P'}_{n}|!P$$

Plutôt que de réduire d'un seul coup une infinité de termes (rappelons que $!P = P|!P$), on utilise n copies, que l'on réduit en n étapes. Cela ne réduit pas la puissance de calcul, mais simplifie la théorie.

Finalement, les règles ne donnent aucune précision sur le *potentiel* de communication d'un processus P avec un autre processus. Connaissant les possibilités de réduction pour P et Q, on ne peut rien dire sur leur mise en parallèle $P|Q$. Robin Milner ne donne pas un système de transitions étiquetées pour résoudre ce problème, mais définit une notion similaire avec une notation tout à fait différente. En fait, il souhaite seulement distinguer les processus capables de communiquer sur le canal α – sujet positif ou négatif – de ceux qui ne le peuvent pas. Aussi, quelques définitions sont ajoutées.

Définition B.2.11
Un processus Q est non gardé *dans P s'il intervient dans P non préfixé.*
P est observable *à α – ce que l'on écrit $P \downarrow_\alpha$ – si $\pi.R$ survient dans P non gardé, où α est le sujet de π et est non restreint (*i.e. *on n'a pas une expression du type $(\nu\ \alpha)\pi.R$).*

Exemple B.2.4
Q est non gardé dans $Q|R$ et $(\nu\ x)Q$ mais est gardé dans $x(y).Q$.
$x(y) \downarrow_x$ et $(\nu\ z)\overline{x}z \downarrow_x$, mais $(\nu\ x)\overline{x}z \not\downarrow_{\overline{x}}$ et $(\nu\ x)(x(y)|\overline{x}z) \not\downarrow_x$ même si cette dernière expression a une réduction.

Une extension intéressante est le π-calcul polyadique qui est l'objet de la section suivante.

B.3 Le π-calcul polyadique

Il est fréquent de vouloir communiquer plusieurs noms à un processus. On pourrait écrire par exemple $\overline{x}yz$ pour envoyer le couple (y, z) sur le canal x. La réception s'écrirait $x(ab)$. Ces écritures seraient les abréviations naturelles respectives de $\overline{x}y.\overline{x}z$

et $x(y).x(z)$. Cependant, cette abréviation donne une fausse idée des réelles communications. Considérons :

$$\overline{x}y_1z_1|\overline{x}y_2z_2|x(yz)$$

le couple de noms (y, z) du processus $x(yz)$ peut être par exemple :

$$(y,z) = (y_1,z_1) \quad (1)$$
$$(y,z) = (y_2,z_2) \quad (2)$$
$$(y,z) = (y_1,y_2) \quad (3)$$
$$(y,z) = (y_2,y_1) \quad (4)$$

Si les égalités (1) et (2) sont bien souhaitées, il n'en est pas de même des égalité (3) et (4). La solution consiste à utiliser un nom privé. Par conséquent, nous adopterons la convention[2] suivante :

Notation B.3.1 (Codage du π-calcul polyadique en π-calcul monadique)

$$\begin{aligned} x(y_1\cdots y_n) &= x(w).w(y_1).\cdots.w(y_n) \\ \overline{x}y_1\cdots y_n &= (\nu\ w)\overline{x}w.\overline{w}y_1.\cdots.\overline{w}y_n \end{aligned}$$

Si $n = 0$, $x = x()$.

Ainsi, dans l'exemple précédent, si on suppose que c'est le premier processus qui communique, on a :

$$\begin{aligned} P \equiv \overline{x}y_1z_1|\overline{x}y_2z_2|x(yz) &= (\nu\ w)\overline{x}w.\overline{w}y_1.\overline{w}z_1|(\nu\ w)\overline{x}w.\overline{w}y_2.\overline{w}z_2|x(w).w(y).w(z) \\ &\overset{1}{\equiv} (\nu\ w_1)\overline{x}w_1.\overline{w_1}y_1.\overline{w_1}z_1|(\nu\ w_2)\overline{x}w_2.\overline{w_2}y_2.\overline{w_2}z_2|x(w).w(y).w(z) \\ &\rightarrow \mathbf{0}|(\nu\ w_2)\overline{x}w_2.\overline{w_2}y_2.\overline{w_2}z_2|\mathbf{0} \equiv Q \end{aligned}$$

Et donc en utilisant la règle STRUCT, on a bien

$$P \rightarrow Q$$

La communication supplémentaire induite ne pose donc aucun problème de sémantique.

Toutefois, Milner propose de rendre les communications polyadiques primitives. La raison est que la notation §B.3.1 p284 permet des écritures du genre

$$\overline{x}y_1\cdots y_n|x(z_1\cdots z_l)$$

et ne précise pas si $n < l$ ou si $n > l$. La notion de sorte – un typage un peu particulier que nous verrons en §B.4 p290 – résoudra cette incohérence. Pour cela, Milner introduit donc la notion d'*abstractions* dans le π-calcul.

[2]Momentanément comme nous allons le voir en §B.3.1 p286...

Définition B.3.1 (Abstraction)
Une abstraction *est une expression de la forme :*

$$(\lambda\ x_1 \cdots x_n)P$$

ou de manière équivalente

$$(\lambda\ x_1) \cdots (\lambda\ x_n)P$$

Ces abstractions sont bien différentes de celles du λ-calcul [52], car ici, les noms liés ne peuvent être instanciés qu'en des noms, jamais en des termes composés. Nous écrirons désormais :

Notation B.3.2

$$\begin{array}{rcl} K(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P & \Leftrightarrow & K \stackrel{\text{def}}{=} (\lambda\ x_1 \cdots x_n)P \\ x(y).P & \stackrel{\text{def}}{=} & x.(\lambda\ y).P \end{array}$$

Nous dirons que, dans le processus $x.(\lambda\ y).P$, x est la *localisation* de l'abstraction $(\lambda\ y).P$. L'abstraction $(\lambda\ y_1 \cdots y_n)P$ est d'arité n. En particulier, un processus est une abstraction d'arité zéro.

De manière symétrique, nous définissons la *concrétion* :

Définition B.3.2 (Concrétion)
Une concrétion est une expression de la forme

$$[x_1 \cdots x_n]P$$

ou de manière équivalente

$$[x_1] \cdots [x_n]P$$

Les préfixes négatifs subissent la même transformation :

Notation B.3.3

$$\overline{x}y_1 \cdots y_n.P \stackrel{\text{def}}{=} \overline{x}.[y_1 \cdots y_n]P$$

Dans le processus $\overline{x}.[y_1 \cdots y_n]P$, $\overline{x}$ est la *co-location* et $[y_1 \cdots y_n]P$ est la *concrétion* d'arité n, équivalente à $[y_1] \cdots [y_n]P$. Tous processus est une concrétion d'arité zéro.

Quelques notations seront simplifieront l'écriture :

Notation B.3.4

$$\begin{aligned} \overline{x}.[\vec{y}].\mathbf{0} &= \overline{x}.[\vec{y}] \\ \overline{x}.[].P &= \overline{x}.P \\ \overline{x}.[].\mathbf{0} &= \bar{x} \end{aligned}$$

D'où la nouvelle syntaxe :

Règles syntaxiques B.3.1 (π-calcul polyadique)

$$\begin{aligned} \text{Processus Normaux} : N &::= \alpha.A \Big| \mathbf{0} \Big| M + N \\ \text{Processus} : P &::= N \Big| P|Q \Big| !P \Big| (\nu\ x)P \\ \text{Abstractions} : F &::= P \Big| (\lambda\ x)F \Big| (\nu\ x)F \\ \text{Concrétions} : C &::= P \Big| [x]C \Big| (\nu\ x)C \\ \text{Agents} : A &::= F \Big| C \end{aligned}$$

Nous devons redéfinir la relation de congruence structurelle présentée en §B.2.9 p280 pour qu'elle prenne en compte la nouvelle syntaxe. En réalité, nous gardons les règles 1 à 6 auxquelles nous ajoutons trois autres règles :

Définition B.3.3 (Congruence structurelle ≡ du π-calcul polyadique)
La congruence structurelle ≡ est la plus petite relation sur $\mathcal{P}$ vérifiant :

1. *des processus sont identiques s'ils ne diffèrent que par un changement de noms liés ;*
2. $(\mathcal{N}/\equiv, +, \mathbf{0})$ *est un monoide symétrique ;*
3. $(\mathcal{P}/\equiv, |, \mathbf{0})$ *est un monoide symétrique ;*
4. $!P \equiv P|!P$
5. $(\nu\ x)\mathbf{0} \equiv \mathbf{0}, (\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$
6. *Si* $x \notin fn(P)$ *alors* $(\nu\ x)(P|Q) \equiv P|(\nu\ x)Q$
7. $(\nu\ y)(\lambda\ x)F \equiv (\lambda\ x)(\nu\ y)F \quad (x \neq y)$
8. $(\nu\ y)[x]C \equiv [x](\nu\ y)C \quad (x \neq y)$
9. $(\nu\ x)(\nu\ y)A \equiv (\nu\ y)(\nu\ x)A$, $(\nu\ x)(\nu\ x) \equiv (\nu\ x)$

Définition B.3.4 (Forme standard)
Toute abstraction ou concrétion est convertible en une forme standard *:*

$$\begin{array}{c} \textit{abstraction : } F \equiv (\lambda\ \vec{x})P \textit{ et} \\ \textit{concrétion : } C \equiv (\nu\ \vec{y})[\vec{x}]P \textit{ avec } (\vec{y} \subseteq \vec{x}) \end{array}$$

L'arité d'une abstraction F – notée $|F|$ – ou d'une concrétion – notée $|C|$ – est la taille du vecteur $\vec{x}$ – noté $|\vec{x}|$ – dans sa forme normale.

La communication s'étend donc au cas polyadique :

$$x(\vec{y}).P|\overline{x}\vec{z}.Q \rightarrow P\{\vec{z}/\vec{y}\}|Q$$

où $\vec{y}$, $\vec{z}$ sont de même taille, les composantes de $\vec{y}$ étant toutes distinctes.

Définition B.3.5 (Pseudo-application)
On appelle pseudo-application *la communication entre une abstraction et une concrétion :*

$$\begin{array}{rcl} F & \equiv & (\lambda\ \vec{x})P \\ C & \equiv & (\nu\ \vec{z})[\vec{y}]Q \\ \vec{x} \cap \vec{z} & = & \emptyset \\ w.F|\overline{w}.C \rightarrow F \bullet C & \stackrel{\text{def}}{=} & (\nu\ \vec{z})(P\{\vec{y}/\vec{x}\}|Q) \end{array}$$

Définition B.3.6 (Relation de réduction)
Notre relation de réduction est donc la plus petite relation satisfaisant les règles suivantes :

$$\text{COMM} : (\cdots + x.F)|(\cdots + \overline{x}.C) \rightarrow F \bullet C$$

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu\ x)P \rightarrow (\nu\ x)P'}$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Notons que la réduction n'est définie que sur les processus et non sur les agents quelconques. De plus, dans COMM, F et C doivent avoir même arité. L'application au sens classique du terme est définie comme suit :

Définition B.3.7 (Application)

$$\begin{array}{rcl} ((\lambda\ x)F)y & \stackrel{\text{def}}{=} & (\nu\ w)(w.(\lambda\ x)F)|\overline{w}.[y]) \\ & \rightarrow & (\nu\ w)(F\{y/x\} \mid \mathbf{0}) \\ & \equiv & F\{y/x\} \end{array}$$

Par conséquent, toute instance d'application peut être éliminée en utilisant la congruence structurelle.

B.3.1 Engagement et congruence

Définition B.3.8 (action,continuation et engagement)
Soit $P = \alpha.A$, un processus atomique normal. Alors α est une action[3] *et A est une* continuation. *$\alpha.A$ sera appelé un* engagement.

P est donc un processus qui peut s'engager en α. Nous allons donc définir une relation $\succ$ entre les processus et les engagements :

$$P \succ \alpha.A$$

ce que l'on prononce "P peut s'engager en α". C'est exactement ce que le système de transition décrit dans les premiers travaux sur le π-calcul [178] décrit, avec une notation différente. Par exemple, à la place de $P \succ \overline{x}.[y]P'$, la transition étiquetée $P \xrightarrow{\overline{x}y} P'$ est utilisée. Nous introduisons maintenant l'action inobservable τ dont nous aurons besoin dans la suite :

Définition B.3.9 (Action inobservable τ)

$$\tau.P \stackrel{\text{def}}{=} (\nu\ x)(x.P|\overline{x});\quad x \notin fn(P)$$

et dorénavant, les variables de noms pourront prendre la valeur τ :

$$\alpha, \beta, \ldots \in \mathcal{X} \cup \{\tau\}$$

De plus, nous définissons la syntaxe suivante :

Règles syntaxiques B.3.2 (Composition d'abstractions ou de concrétions)

$$F \equiv (\lambda\ \vec{x})P;\ G \equiv (\lambda\ \vec{y})Q$$

$$\vec{x} \notin nm(G);\ \vec{y} \notin nm(P)$$

$$F|G \stackrel{\text{def}}{=} (\lambda\ \vec{x}\vec{y})(P|Q)$$

De même :

$$C \equiv (\nu\ \vec{x})[\vec{u}]P;\ D \equiv (\nu\ \vec{y})[\vec{v}]Q$$

$$\vec{x} \notin D;\ \vec{y} \notin nm(C)$$

$$C|D \stackrel{\text{def}}{=} (\nu\ \vec{x}\vec{y})[\vec{u}\vec{v}](P|Q)$$

[3]En fait, le terme *localisation* d'une action serait plus juste, mais nous emploierons indifféremment l'un où l'autre.

La relation $|$ est associative vis à vis de $\equiv$, mais pas commutative en général, bien qu'elle le soit sur les processus. En effet, on a clairement, dans le cas général, $F|G \not\equiv G|F$: bien que $|\vec{x}\vec{y}| = |\vec{y}\vec{x}|$, la réception de $\vec{x}$ suivi de $\vec{y}$ n'est clairement pas la même chose que la réception de $\vec{y}$ suivi de $\vec{x}$. Par ailleurs, $A|P$ est défini pour n'importe quel agent A et n'importe quel processus P (un processus est à la fois une abstraction et une concrétion d'arité zéro). De plus, $A|P \equiv P|A$. La relation d'engagement est définie comme suit :

Définition B.3.10 (Relation d'engagement)
La relation d'engagement *entre processus et engagements est la plus petite relation satisfaisant les règles suivantes:*

$$\text{SUM} : \cdots + \alpha.A \succ \alpha.A$$

$$\text{COMM} : \frac{P \succ x.F \quad Q \succ \overline{x}.C}{P|Q \succ \tau.(F \bullet C)}$$

$$\text{PAR} : \frac{P \succ \alpha.A}{P|Q \succ \alpha.(A|Q)}$$

$$\text{RES} : \frac{P \succ \alpha.A}{(\nu\ x)P \succ \alpha.(\nu\ x)A}(\alpha \notin \{x, \overline{x}\})$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \succ \alpha.A \quad A \equiv B}{Q \succ \alpha.B}$$

Enfin, avant de donner la définition de la bisimulation, nous introduisons une nouvelle définition :

Définition B.3.11 (Respectabilité)
Soit $|\equiv$ une relation binaire quelconque sur les agents. $|\equiv$ est respectable *si elle inclut la congruence structurelle $\equiv$, et si elle est respectée par la décomposition des concrétions et par l'application des abstractions,* i.e. *:*

- *si $C \equiv (\nu\ \vec{x})[\vec{y}]P$ et $D \equiv (\nu\ \vec{x})[\vec{y}]Q$ et $C \models D$ alors $P \models Q$*
- *si $F \models G$ alors $|F| = |G| = n$ et $\forall \vec{y}, |\vec{y}| = n, F\vec{y} \models G\vec{y}$.*

Définition B.3.12 (Simulation, bisimulation et bisimilarité)
Une relation $\models$ sur les agents est une simulation forte *si elle est respectable et si $P \models Q$ et $P \succ \alpha.A$ alors $\exists B / Q \succ \alpha.B$ et $A \models B$. $\models$ est une* bisimulation forte *si $\models$ et sa réciproque sont des simulations fortes. La* forte bisimilarité *– noté $\dot{\sim}$ – est la plus large bisimulation forte.*

Aussi, pour prouver que $P \dot{\sim} Q$, il suffit de trouver une bisimulation forte qui contient (P, Q) puisque la forte bisimilarité est l'union de toutes les bisimulations fortes. Cependant, la forte bisimilarité n'est pas une congruence : elle n'est pas préservée par substitution de noms.

Exemple B.3.1 (Bisimilarité préservée par substitution de noms)

$$\overline{x}|y \dot{\sim} \overline{x}.y + y.\overline{x}$$

Car,

$$\begin{aligned} \overline{x}|y &\succ \overline{x}.(\mathbf{0}|y) \\ &\succ y.(\overline{x}|\mathbf{0}) \\ \overline{x}.y + y.\overline{x} &\succ \overline{x}.y \\ &\succ y.\overline{x} \end{aligned}$$

Donc :

$$\mathcal{R} = \equiv \cup\{(\overline{x}.(\mathbf{0}|y); \overline{x}.y)\} \cup \{(y.(\overline{x}|\mathbf{0}); y.\overline{x}\} \cup \{(\overline{x}|y; \overline{x}.y + y.\overline{x})\}$$

est une bisimulation. En effet, $\mathcal{R}$ est respectable (évident) et on a bien :

$$P \equiv \overline{x}|y \; \mathcal{R} \; \overline{x}.y + y.\overline{x} \equiv Q$$

par construction, et

$$(P \succ \alpha.A \Rightarrow \exists B / Q \succ \alpha.B; A \; \mathcal{R} \; B)$$

Exemple B.3.2 (Bisimilarité non préservée par substitution de noms)
Par contre, pour $\sigma = y/x$,

$$P\sigma \equiv \overline{x}|x \not\dot{\sim} \overline{x}.x + x.\overline{x} \equiv Q\sigma$$

P peut en effet s'engager en $\tau.\mathbf{0}$ ce que ne peut faire Q.

Nous définissons donc :

Définition B.3.13 (Congruence forte)
P et Q sont fortement congruent *– et l'on écrit $P \sim Q$ – si $P\sigma \dot{\sim} Q\sigma$ pour toute substitution σ.*

B.4 Sortes

Les sortes sont les équivalents en π-calcul du typage du λ-Calcul.

Définition B.4.1 (Sorte de sujet)
Soit $I \subset \mathbb{N}$ et $\mathcal{S} = \{S_i, i \in I\}$. On appelle une sorte de sujet *un élément $S \in \mathcal{S}$. S peut être un nom quelconque qui n'est pas dans $\mathcal{X}$.*

Pour chaque sorte de sujets, il existe une infinité de noms $x \in \mathcal{X}$ tel que x soit de sorte S – on note $x : S$.

Définition B.4.2 (Sorte d'objet)
Une sorte d'objet *est une séquence sur* $\mathcal{S}$ *:*

$$Ob(\mathcal{S}) = \mathcal{S}^*$$

Nous écrirons $(S_1, \ldots, S_n)$ *pour une sorte d'objet. La sorte d'objet vide est* $()$*. On note* $\hat{}$ *l'opération de concaténation de sortes d'objets:*

$$(S_1, \ldots, S_n)\hat{}(S'_1, \ldots, S'_m) = (S_1, \ldots, S_n, S'_1, \ldots, S'_m)$$

Définition B.4.3 (Typage)
Un typage *sur* $\mathcal{S}$ *est une fonction partielle:*

$$ob : \mathcal{S} \rightharpoonup Ob(\mathcal{S})$$

Si ob est finie, on écrira $ob = \{S_1 \mapsto ob(S_1), \ldots, S_n \mapsto ob(S_n)\}$.

Un typage détermine pour tout nom $x \in \mathcal{X}$ la sorte d'objet qu'il peut transporter.

Définition B.4.4 (Respect de typage)
Un agent A respecte *un typage ob, on dit aussi est* bien typé *pour ob, si on peut trouver une sorte d'objet* $s \in Ob(\mathcal{S})$ *telle que* $A : s$ *par les règles suivantes :*

$$\mathbf{0} : () \quad \frac{x : S \quad F : ob(S)}{x.F : ()}$$

$$\frac{x : S \quad C : ob(S)}{\overline{x}.C : ()} \quad \frac{P : ()}{\tau.P : ()}$$

$$\frac{M : () \quad N : ()}{M + N : ()} \quad \frac{P : () \quad Q : ()}{P|Q : ()}$$

$$\frac{P : ()}{!P : ()} \quad \frac{A : s}{(\nu\ x)A : s}$$

$$\frac{x : S \quad F : s}{(\lambda\ x)F : (S)\hat{}s} \quad \frac{x : S \quad C : s}{[x]C : (S)\hat{}s}$$

Nous pourrions montrer que :

$$\text{Si } x : S, y : S, A : s \text{ alors } A\{y/x\} : s$$

$$\text{Si } A : s, A \equiv B, \text{ alors } B : s$$

$$\frac{F : (S)\hat{}s \quad y : S}{Fy : s}$$

$$\frac{F : s \quad G : t}{F|G : s\hat{}t}$$

Ainsi, le typage $\{NAME \mapsto ()\}$ – une sorte d'objet ne transporte rien – correspond à CCS et $\{NAME \mapsto (NAME)\}$ correspond au π-calcul monadique. On peut remarquer que le codage du π-calcul polyadique en π-calcul monadique vu en §B.3.1 p284 ne respecte aucun typage :

$$x(y_1 \cdots y_n) = x(w).w(y_1).\cdots .w(y_n)$$

Le nom x ne peut transporter à la fois un vecteur $\vec{y}$ d'arité n et un nom w lui-même transportant un nom y_i, $i \in 1, \ldots, n$. C'est l'une des raisons qu'invoque Milner pour justifier l'introduction de la polyadicité.

B.5 Le π-calcul d'ordre supérieur

L'idée est de permettre des écritures du genre :

$$P \stackrel{\text{def}}{=} \overline{x}.[R]P' \text{ et } Q \stackrel{\text{def}}{=} x.(\lambda\ X)(X|Q')$$

Le processus P envoie un *processus* R via le nom x et Q, le recevant, l'exécute en parallèle avec Q'. On aimerait donc avoir la réduction :

$$P|Q \rightarrow P'|R|Q'$$

Cette notation n'est pas autorisée dans le π-calcul présenté jusqu'à présent. En effet, seul le passage de noms est admis. Toutefois, considérons le codage suivant :

$$\hat{P} \stackrel{\text{def}}{=} \overline{x}.(\nu\ z)[z](z.R|P')$$

$$\hat{Q} \stackrel{\text{def}}{=} x.(\lambda\ z)(\overline{z}|Q')$$

On a alors :

$$\begin{array}{rcl} \hat{P}|\hat{Q} & \equiv & \overline{x}.(\nu\ z)[z](z.R|P')|x.(\lambda\ z)(\overline{z}|Q') \\ & \rightarrow & (\nu\ z)((z.R|P')|(\overline{z}|Q')) \\ & \rightarrow & P'|R|Q' \end{array}$$

On aimerait donc avoir pour tout processus d'ordre supérieur P et Q,

$$P \sim Q \Longleftrightarrow \hat{P} \sim \hat{Q}$$

pour une congruence naturelle $\sim$.

Cette propriété n'est pas vérifiée pour certaines congruences pour des raisons qui sortent du cadre de ce document [204]. Milner introduit donc l'ordre supérieur directement dans la syntaxe :

Notation B.5.1 (Variables d'abstractions)
On note $X, Y, \ldots$ les variables d'abstractions.

La nouvelle syntaxe devient :

Règles syntaxiques B.5.1

$$\begin{array}{rcl}
\text{Processus Normaux} : N & ::= & \alpha.A \Big| \mathbf{0} \Big| M + N \\
\text{Processus} : P & ::= & N \Big| P|Q \Big| !P \Big| (\nu\ x)P \Big| F \\
\text{Abstractions} : F & ::= & P \Big| (\lambda\ x)F \Big| (\lambda\ X)F \Big| (\nu\ x)F \Big| X \Big| Fx \Big| FG \\
\text{Concrétions} : C & ::= & P \Big| [x]C \Big| [F]C \Big| (\nu\ x)C \\
\text{Agents} : A & ::= & F \Big| C
\end{array}$$

Milner montre que cette extension n'engendre pas de problème particulier (si ce n'est quelques règles ou définitions supplémentaires). Nous ne détaillerons le π-calcul d'ordre supérieur et terminerons simplement notre introduction avec les sortes d'ordre supérieur.

Milner introduit la notion de *sorte de données* :

Définition B.5.1 (Sorte de données)

$$Dat(\mathcal{S}) \stackrel{\text{def}}{=} \mathcal{S} \cup Ob(\mathcal{S})$$
$$Ob(\mathcal{S}) \stackrel{\text{def}}{=} Dat(\mathcal{S})^*$$

Ainsi, une sorte d'objet peut être : $(S_1(S_2S_1)S_3)$.

Définition B.5.2 (Règles de typage)
On ajoute à la définition §B.4.4 p291 les règles suivantes :

$$\frac{X : S \quad F : t}{(\lambda\ X)F : (s)\hat{}t}$$

$$\frac{F : (S)\hat{}t \quad x : S}{Fx : t} \qquad \frac{F : (s)\hat{}t \quad G : s}{FG : t}$$

$$\frac{F : s \quad C : t}{[F]C : (s)\hat{}t}$$

Notre introduction s'achève ici.

Annexe C

Modélisation d'un système d'agents mobiles

Le domaine des agents mobiles a très vite été confronté à des problèmes de modélisation. Si dans un premier temps, l'infrastructure nécessaire au déploiement d'agents mobiles a été étudiée dans un cadre général [126], il est apparu nécessaire de donner des fondements méthodologiques et théoriques.

Des fondements méthodologiques ont été proposés par les concepteurs du système d'agents mobiles *Aglets* [53] étudié en §4.3.2 p47 dans un catalogue de patterns [19] spécifique au domaine et similaire au très connu *design patterns* [82]. Ces patterns définissent des solutions génie logiciel à des problèmes fréquemment rencontrés dans le domaine des agents mobiles tels que les notions de *voyage*, de *tâche* et d'*interaction*.

Les fondements théoriques sont basés sur des modélisations comportementales ou structurelles des systèmes d'agents mobiles.

C.1 Modélisation structurelle

Une modélisation structurelle des agents mobiles a été proposée en [187]. Elle propose un ensemble de diagrammes UML [159] qui décrivent l'architecture et le comportement des entités impliqués dans un système d'agents mobiles. Cette architecture est ensuite adaptée à différents systèmes réels tel que AgentSpace [61], les aglets [53] ou Telescript [132]. L'architecture proposée est représentée sur la figure Fig. C.1 p296. Les principales entités de cette architecture sont :

- **le client :** il manipule l'agent à travers une référence de type `AgentView` ; il peut être un autre agent ou un autre objet Java ;
- **l'image d'un agent** : la classe `AgentView` est une adaptation de la notion de

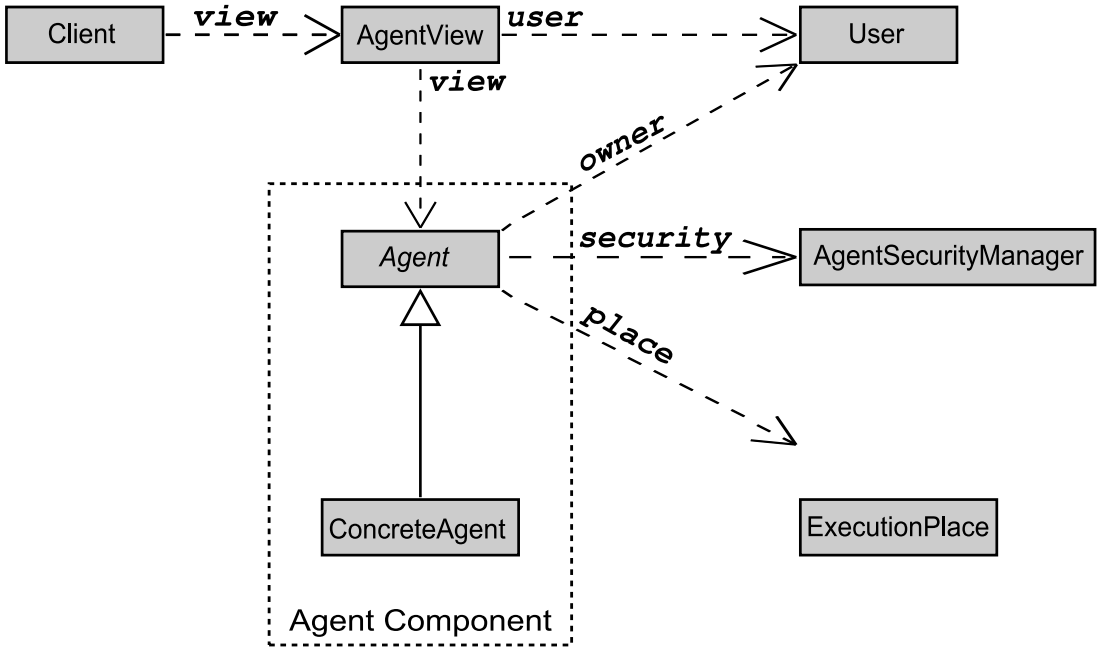


FIG. C.1 – Architecture d'un système d'agents mobiles.

proxy (*cf.* §9.1 p79) et permet d'assurer la transparence des appels distants, de la localisation et la protection de son agent associé ;

- **l'utilisateur :** associé à un identificateur unique est représenté par les instances de la classe `User` qui peut contenir par exemple le nom, la clé publique, et les certificats de l'utilisateur ;
- **l'agent :** représenté par la classe abstraite `Agent` qui contient trois groupes de méthodes : les méthodes publiques non redéfinissable (`moveTo()`, `sendMessage()`, `clone()`), des méthodes évenementielles (onCreation(), handleMessage()) et des méthodes métiers, généralement privés, définient dans les sous-classes et utilisés par les méthodes évenementielles ; un agent fournit des services de manière transparente tel que la persistance, la communication, la mobilité ou le nommage.
- **l'agent métier :** représenté par la classe `ConcreteAgent`, une sous-classe de `Agent` et (re-)définit les méthodes évenementielles ainsi que les méthodes métiers ; c'est évidemment cette classe qui définit le comportement de l'agent ;
- **le gestionnaire de sécurité :** représenté par la classe `SecurityManager` définit les politiques d'accès à l'agent ; il contrôle les opérations accessibles via l'image `AgentView` ;

• **la localisation :** définit par la classe `ExecutionPlace` représente l'environnement d'exécution de l'agent.

Cette architecture est suffisamment générale pour être adaptée à différentes implémentations de système d'agents mobiles. Cependant, si la modélisation UML apporte un avantage certain dans un développement collaboratif (pour communiquer entre autres), il permet difficilement de réaliser des preuves formelles sur le comportement d'un système d'agents.

C.2 Modélisation comportementale

Le comportement d'un système d'agents mobiles est assez complexe à étudier en raison du caractère très volatile des entités mises en œuvre. La modélisation comportementale proposée en [185] décompose le modèle en trois niveaux :

• **le niveau système** décrit le système dans son ensemble et les interactions entre les machines du système ; il est représenté par un graphe orienté étiqueté ;

• **le niveau machine** décrit les interactions entre agents (d'une même machine ou de machines distinctes) ; il est modélisé par un réseau de pétri étendu [224] ;

• **le niveau agent** décrit le comportement d'un agent ; il est modélisé par une machine à état fini.

Cette approche permet donc une modélisation de très haut niveau d'un système d'agents mobiles. Ce modèle n'est cependant pas assez proche des implémentations existantes. Il ne rend pas compte des problèmes de sécurité liés au support d'exécution par exemple.

C.3 Proposition de notre propre modélisation

Dans ce chapitre, nous proposons une modélisation en π-calcul d'un système d'agents mobiles qui reflète les systèmes d'agents étudiés précédemment.

C.3.1 Modèle informel

Une représentation intuitive d'un système d'agents est schématisée sur la figure Fig. C.2 p298. Sur celle-ci, on distingue plusieurs supports d'exécution – que nous appellerons *serveurs d'agents* dans la suite – et des agents. Le serveur S_1 *contient* les agents : A_1 et A_2, le serveur S_2, les agents A_3, A_4 et A_5 et le serveur S_3 l'agent A_6. A_3 et A_6 ont migré respectivement de S_1 à S_2 et de $S2$ à S_3.

Cette relation de *contenance* a son importance. Considérons en effet maintenant le système tout entier, sans se préoccuper de la notion d'intérieur d'un serveur

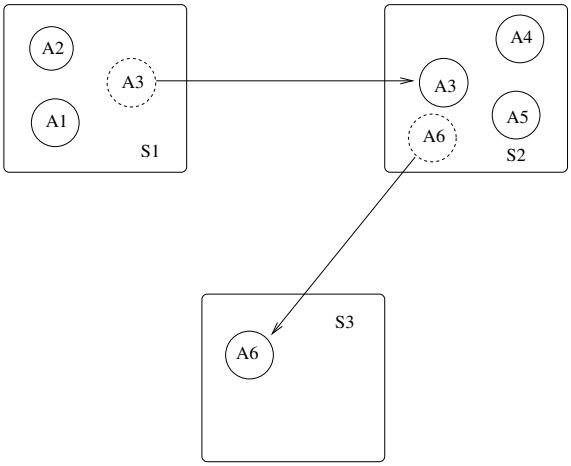


FIG. C.2 – Représentation de *l'intérieur* des serveurs.

d'agents. Nous avons donc un ensemble d'éléments de types différents (agents et serveurs) qui s'exécutent en *concurrence*. Bien entendu, nous pourrions représenter un réseau de serveurs d'agents par un graphe. Néanmoins, si celui-ci donne bien des informations suffisantes sur les connections entre les serveurs, comment introduire la notion de code mobile ? L'étiquetage, statique, ne rend pas compte des données qui transitent. Dans la modélisation d'un réseau classique, cela n'a pas d'importance, les données ne sont généralement pas représentées. Dans notre cas, il n'en est pas de même, puisque notre modèle doit rendre compte de l'activité des agents. Il faut donc représenter de manière simple la notion d'intérieur de serveur.

Lorsqu'un agent est contenu dans un serveur, nous dirons qu'il lui est *local*. Dans ce cas, le serveur a un accès privilégié : il peut lui interdire la migration, lui interdire l'allocation de ressources, etc. Si l'agent migre, ce privilège passe d'un serveur à un autre. L'agent n'est plus local au serveur qu'il quitte. Cette notion de privilège est schématisée sur la figure FIG. C.3 p299. L'agent A_3 migrant de S_1 à S_2 change son lien privilégié de S_1 à S_2. C'est cette notion que nous allons formaliser à la place de la notion d'intérieur d'un serveur d'agents.

L'idée est donc de prendre en compte la localité d'un agent. Cette notion de localité doit être éclaircie. L'idée consiste à restreindre la *vue* des entités du réseau. La vue d'une entité est l'ensemble des entités qui lui sont accessibles[1]. Évidemment, une entité appartient à sa propre vue.

Le *serveur local* d'un agent est la seule entité du réseau dont la vue contient l'agent. Il est la *localisation* de l'agent. Cette restriction implique que :

- seul le serveur local peut appeler une méthode de l'agent ;
- la communication directe entre agents est interdite.

[1]Une entité a accès à une autre si elle possède une référence directe sur elle.

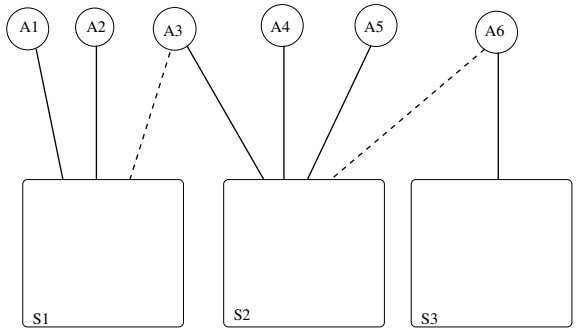


Fig. C.3 – Représentation par restriction de la *vue* des entités.

La dernière de ces conséquences peut sembler trop restrictive. En effet, si les agents ne peuvent communiquer entre eux, l'intérêt d'un tel modèle paraît limité. Cela dit, rien n'interdit un référencement indirect (utilisation d'un *proxy* (*cf.* §9.1 p79 par exemple). Nous appellerons identificateur local – et nous noterons *id* – une telle référence indirecte. Un *id* devra être unique pour chaque agent d'un serveur et être fourni à toute entité souhaitant communiquer avec un agent. Nous pourrons donc considérer que l'*id* d'un agent est la seule référence valide dans le réseau.

Par ailleurs, il semble incohérent de considérer un agent hors d'un système d'agents. Un agent existe par sa présence en tant que "visiteur" d'un serveur. Aussi, seul un serveur est habilité à transformer un objet en agent.

C.3.2 Choix d'un formalisme

La modélisation de notre architecture nécessite de choisir un formalisme. Il existe plusieurs modèles de calculs de processus concurrents [157]. Leur précurseur est CCS (Calculus of Concurrent Processses) [149]. En annexe §A p273 nous donnons une courte description des différents modèles de calculs disponibles et les raisons qui nous ont amenées à choisir le π-calcul pour notre modélisation.

Dans la suite, nous supposerons que le π-calcul n'est pas complètement étranger au lecteur. Une introduction au π-calcul est donné en annexe §B p277.

C.3.3 Formalisation

Avant d'entrer dans les détails du formalisme, rappelons brièvement les caractéristiques de notre modélisation :

Définition C.3.1 (Caractéristiques du modèle)

1. *nous représentons un ensemble de systèmes d'agents s'exécutant en concurrence ;*

2. *la notion d'intérieur d'un système d'agents est représentée par la restriction de la vue des autres objets du réseau : aucune référence directe n'est autorisée entre deux agents. Il faut utiliser un identificateur local valide pour faire référence à un agent du réseau.*

En prenant en compte les règles §1 p299 et §2 p300, un réseau $\mathcal{R} = S_1, \ldots, S_n$ où S_i est un système d'agents contenant k_j agents $A_{i,1}, \ldots, A_{i,k_j}$ s'écrit en π-calcul :

$$\begin{array}{rcl} S_1 & | & \{A_{1,1}|A_{1,2}|\cdots|A_{1,k_1}\}\,| \\ S_2 & | & \{A_{2,1}|A_{2,2}|\cdots|A_{2,k_2}\}\,| \\ \vdots & | & \vdots \\ S_n & | & \{A_{n,1}|A_{n,2}|\cdots|A_{n,k_n}\} \end{array}$$

Dans notre modèle à base d'objets nous factoriserons les noms de méthodes :

Notation C.3.1 (Factorisation des noms de méthodes)
Soit $\{\widetilde{X}\#x_1, \ldots, \widetilde{X}\#x_n\}$ un ensemble de noms de méthodes. On notera $\widetilde{X}$ cet ensemble[2]*. On écrira $\widetilde{X} \asymp \{x_1, \ldots, x_n\}$ pour définir $\widetilde{X}$. Par soucis de clarté, nous écrirons x_i pour faire référence à la méthode $\widetilde{X}\#x_i$ lorsque $\widetilde{X}$ sera évident. Enfin,*

$$\overline{\widetilde{X}\#x_i} = \widetilde{X}\#\overline{x_i}$$

Observons sur un exemple comment le π-calcul permet une modélisation intuitive de code Java.

Exemple C.3.1 (Modélisation de code Java en π-calcul)
Considérons le fragment de code §C.1 p301. Cette classe peut-être formalisée ainsi :

[2] On peut considérer le symbole # comme l'opérateur de concaténation.

```
1  public class Classe{
2          public int ma(int xa){
3                  return plus(xa,2);
4          }
5          public type mb(type xb){
6                  ...
7          }
8          .
9          .
10         .
11         public type mz(type xz){
12                 ...
13         }
14 }
```

Prog. C.1 – Modélisation d'une classe Java en π-calcul.

$$
\begin{aligned}
&\widetilde{m} \asymp \{ma, mb, \ldots, mz\}\\
&Classe \equiv (\lambda\ \widetilde{m})\Big\{\\
&\qquad \widetilde{m}\#ma(\lambda\ xa, resultat).\\
&\qquad\qquad \overline{plus}[xa, 2, resultat]\\
&\qquad +\\
&\qquad \widetilde{m}\#mb(\lambda\ xb, resultat).\\
&\qquad\qquad \ldots\\
&\qquad +\\
&\qquad \vdots\\
&\qquad +\\
&\qquad \widetilde{m}\#mz(\lambda\ xz, resultat).\\
&\qquad\qquad \ldots\\
&\Big\}
\end{aligned}
$$

Le code:

```
Classe o = new Classe();
int y = plus(o.ma(3), 2);
```

est formalisé par l'expression :

$$(\nu\ \widetilde{m})\Big\{\ (Classe)(\widetilde{m})\ |\ (\nu\ y, res)\ \Big[\ \widetilde{m}\#\overline{ma}[3, res]\ |\ \overline{plus}[res, 2, y]\Big]\Big\}$$

L'accès à une variable ou une méthode est modélisé par l'accès à un nom et nécessite donc une communication. On peut en effet considérer que l'accès à la mémoire (variable ou code) est une communication. Par conséquent, l'accès à la variable y dans la suite du processus présenté nécessitera sa mise en parallèle avec l'expression :

$$y(\lambda x)$$

C.3.3.1 Conventions de nommage

Un serveur est un ensemble de services accessibles depuis l'extérieur et un ensemble d'agents accessibles seulement par leur *id* respectif.

Dans l'ensemble des systèmes d'agents présentés, le support d'exécution est l'élément central :

- il contient les agents ;
- par lui, sont expédiés ou reçus les agents ;
- il crée un agent[3].

Nous allons donc fournir à notre serveur d'agents les services nécessaires :

$$createAgent, sendAgent, receiveAgent, killAgent$$

Deux noms nous seront nécessaires pour la gestion de l'ensemble des agents :

$$NewAgent, DeleteAgent.$$

Ils seront internes au serveur et ne seront pas accessibles aux autres objets du réseau. Ils permettront respectivement de créer ou de supprimer un agent.

Dans les systèmes précédemment étudiés, un agent dispose de méthodes qui lui permettent, entre autres, de gérer son espace de données. Nous avons vu qu'un mécanisme de type événementiel est généralement proposé. Nous allons nous conformer à ce mécanisme en définissant[4] l'ensemble des méthodes suivantes :

`onCreation()` : invoquée après la création de l'agent ;

`onMigrating()` : invoquée avant la migration ;

`onMigration()` : invoquée après la migration ;

`onDisposing()` : invoquée après la destruction de l'agent.

[3]Dans Voyager, la facette IAgent peut être attachée à n'importe quel objet (sérialisable). Un objet n'ayant pas cette facette n'est pas un agent. On peut donc considérer que la création de l'agent est liée non pas à une instantiation d'objet (comme dans tous les autres systèmes) mais à l'ajout de cette facette. De ce point de vue, c'est bien le support d'exécution qui "crée l'agent".

[4]Très inspiré par le système des aglets...

Par contre, nous allons séparer le code de ces méthodes – que l'agent est le seul à connaître a priori – de leur noms, connus aussi du serveur (il doit les invoquer). Ainsi, un agent est défini par un ensemble d'abstractions

$$\{C, R, M_o, M_i, D, Extra\}$$

(le code), et par un ensemble de noms (de méthodes)

$$\{onCreation, onMigration, onMigrating, onDisposing, migrate, dispose\}$$

C est l'abstraction qui correspond à la création de l'agent : elle doit effectuer des tâches spécifiques à sa création après avoir reçu en paramètre un argument "utilisateur"[5], son *id* et le serveur créateur ($|C| = 3$). De même, R ($|R| = 0$) est l'action de l'agent, M_o ($|M_o| = 1$) est associée à une action effectuée lorsque l'agent a migré, le nouveau serveur hôte étant passé en argument, M_i ($|M_i| = 1$) est associée à l'action à effectuer juste avant de migrer, le serveur destination lui est passé en argument, et enfin, D ($|D| = 0$) est associée à l'action à effectuer juste avant la mort de l'agent.

L'abstraction $Extra$ est un peu particulière. Cette abstraction d'arité quelconque (*cf.* §C.3.4.4 p311) ne sert pas au serveur. Elle représente tout ce que l'agent est capable de faire et que le serveur ne connaît pas. Par exemple, un agent pourrait migrer vers un serveur particulier, calculer une somme, revenir et proposer cette somme aux agents qui le souhaitent. Le calcul de la somme est effectué par R mais l'enregistrement du résultat se trouve sous forme d'une expression en π-calcul dans $Extra$. Dans la suite, nous ne considérerons plus $Extra$ sauf mention particulière[6].

Nous l'avons vu, les abstractions représentent le code des méthodes de l'agent. Il leur est associé un nom (sauf pour R). La correspondance entre nom et abstraction est donnée dans le tableau Tab. C.1 p303.

Abstraction	**Nom**
C	$onCreation$
M_i	$onMigrating$
M_o	$onMigration$
D	$onDisposing$

Tab. C.1 – Correspondance entre noms et abstractions

Deux autres méthodes sont définies pour l'agent :

`migrate()` : invoquée pour une demande de migration ;

`dispose()` : invoquée pour une demande de terminaison.

[5]Nous verrons en §C.3.4.4 p311 que cet argument est un peu particulier.

[6]Nous verrons en §C.3.4.1 p309 que $Extra$ représente entre autres, l'état de l'agent.

Les noms de ces méthodes ne sont pas associés à des abstractions. Elles sont externes, accessibles à la fois par l'agent et par tout autre objet du réseau connaissant son *id*.

Nous pouvons donc définir les deux objets de notre système d'agents :

Définition C.3.2
Un serveur est un ensemble de noms

$$\widetilde{S} \asymp \{createAgent, sendAgent, receiveAgent, killAgent\}$$

Un agent est composé de deux parties :

$$\widetilde{AgentCode} \asymp \{C, R, M_o, M_i, D, Extra\}$$

$$\begin{aligned}\widetilde{AgentMethod} \asymp \{&onCreation,\\ &onMigration,\\ &onMigrating,\\ &onDisposing,\\ &migrate,\\ &dispose\}\end{aligned}$$

Nous allons maintenant proposer dans l'ordre : la modélisation du serveur d'agent, la modélisation des services qu'il offre, et enfin, la modélisation d'un agent proprement dite.

C.3.3.2 Modélisation du serveur

On définit la création d'un serveur par :

$$\begin{aligned}Server \equiv\ &(\lambda\ \widetilde{S})\\ &(\nu\ NewAgent, DeleteAgent)\\ &\quad !((Agents)(\widetilde{S}, NewAgent, DeleteAgent)\\ &\quad\ |(Services)(\widetilde{S}, NewAgent, DeleteAgent))\end{aligned}$$

Après avoir créé deux noms $NewAgent$ et $DeleteAgent$, l'abstraction $Server$ applique les deux abstractions $Agents$ et $Services$. L'opérateur de réplication permet l'attente infinie d'une requête. On crée donc un serveur de la façon suivante :

$$(\nu\ \widetilde{S})(Server)(\widetilde{S})$$

Agents est défini comme suit :

$$
\begin{array}{l}
Agents \equiv (\lambda\ \widetilde{S}, NewAgent, DeleteAgent).\Big\{ \\
\quad NewAgent(\lambda\ id, \widetilde{AgentCode}). \\
\quad\quad (\nu\ \widetilde{AgentMethod}) \\
\quad\quad\quad \Big\{(Agent)(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S}) \\
\quad\quad\quad\ |!\overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}]\Big\} \\
\quad + \\
\quad\quad DeleteAgent(\lambda\ id). \\
\quad\quad\quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
\quad\quad\quad \widetilde{AgentMethod}\#\overline{onDisposing} \\
\Big\}
\end{array}
$$

La création d'un objet est effectuée via le nom $NewAgent$ qui reçoit un id nouvellement créé et le code de l'agent. La création de l'agent proprement dite est réalisée par la création d'un ensemble de noms $\widetilde{AgentMethod}$ et par l'application de l'abstraction $Agent$ (d'arité quatre) et du quadruplet :

$$(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S})$$

Ainsi, l'agent connaît :

- son code ;
- l'ensemble $\widetilde{AgentMethod}$ de ses noms de méthodes ;
- son identificateur id sur ce serveur ;
- les méthodes $\widetilde{S}$ du serveur sur lequel il réside : c'est son serveur local.

La création s'achève avec la mise à disposition du code et du nom des méthodes de l'agent à tous objets connaissant son *id*. Il peut paraître étonnant de laisser disponible à la fois le code et les méthodes de l'agent. Cette problématique sera discutée une fois le modèle entièrement formalisé en §C.3.4.2 p309.

La destruction d'un agent est triviale. On récupère le code et les méthodes de l'agent à partir de l'*id*. Elle nécessite l'emploi du nom de méthode *onDisposing* afin de donner à l'agent la possibilité de faire une dernière tâche. Nous verrons malgré tout que cette destruction est problématique.

C.3.3.3 Modélisation des services

Les services représentent l'ensemble des noms de méthodes publiques (accessibles à quiconque). L'une de ces méthodes est *createAgent*. Elle reçoit le code de l'agent

à créer, construit un nouvel identificateur, fait un appel à $NewAgent$ pour créer effectivement l'agent, exécute le code C de cet agent (via sa méthode $onCreation$), et finalement retourne l'identificateur du nouvel agent pour permettre aux objets extérieurs de communiquer avec lui.

Nous aurons donc l'expression :

$$\begin{array}{l}\widetilde{S}\#createAgent(\lambda\ \widetilde{AgentCode}, arg, returnid).\\ \quad (\nu\ id)\overline{NewAgent}[id, \widetilde{AgentCode}].\\ \quad\quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}).\\ \quad\quad \widetilde{AgentMethod}\#\overline{onCreation}[arg].\\ \quad\quad \overline{returnid}[id]\end{array}$$

De même, l'envoi d'un agent par un serveur $\widetilde{S}$ sur un serveur $\widetilde{S}'$ passe par plusieurs étapes :

- recevoir l'identificateur de l'agent à envoyer par l'intermédiaire du nom de méthode $\widetilde{S}\#sendAgent$;
- appeler la méthode $onMigrating$ de l'agent en question ;
- appeler la méthode $\widetilde{S}'\#receiveAgent$ du serveur récepteur ;
- renvoyer l'id de l'agent dans son nouveau serveur local et le rendre inaccessible dans l'ancien.

Par conséquent, nous aurons :

$$\begin{array}{l}\widetilde{S}\#sendAgent(\lambda\ id, \widetilde{S}', returnNewid).\\ \quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}).\\ \quad \widetilde{AgentMethod}\#\overline{onMigrating}[\widetilde{S}'].\\ \quad (\nu\ returnid)\widetilde{S}'\#\overline{receiveAgent}[\widetilde{AgentCode}, returnid].\\ \quad\quad returnid(\lambda Newid).\\ \quad\quad \overline{returnNewid}[Newid]\end{array}$$

Les deux autres méthodes $receiveAgent$ et $killAgent$ sont similaires. Voici la

modélisation complète du serveur :

$$
\begin{aligned}
Server \equiv (\lambda\ \widetilde{S}).\Big\{ & (\nu\ NewAgent, DeleteAgent) \\
& !((Agents)(\widetilde{S}, NewAgent, DeleteAgent) \\
& \ |(Services)(\widetilde{S}, NewAgent, DeleteAgent)) \Big\} \\
Agents \equiv (\lambda\ \widetilde{S}, NewAgent, DeleteAgent).\Big\{ & \\
& NewAgent(\lambda\ id, \widetilde{AgentCode}). \\
& \quad (\nu\ \widetilde{AgentMethod}) \\
& \quad\quad \Big\{(Agent)(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S}) \\
& \quad\quad\ |!\overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}]\Big\} \\
& + \quad DeleteAgent(\lambda\ id). \\
& \quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
& \quad \widetilde{AgentMethod}\#\overline{onDisposing}\ \Big\} \\
Services \equiv (\lambda\ \widetilde{S}).\Big\{ & \\
& \widetilde{S}\#createAgent(\lambda\ \widetilde{AgentCode}, arg, returnid). \\
& \quad (\nu\ id)\overline{NewAgent}[id, \widetilde{AgentCode}]. \\
& \quad\quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
& \quad\quad \widetilde{AgentMethod}\#\overline{onCreation}[arg]. \\
& \quad\quad \overline{returnid}[id] \\
& + \quad \widetilde{S}\#sendAgent(\lambda\ id, \widetilde{S}', returnNewid). \\
& \quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
& \quad \widetilde{AgentMethod}\#\overline{onMigrating}[\widetilde{S}']. \\
& \quad (\nu\ returnid)\widetilde{S}'\#\overline{receiveAgent}[\widetilde{AgentCode}, returnid]. \\
& \quad\quad returnid(\lambda Newid). \\
& \quad\quad \overline{returnNewid}[Newid] \\
& + \quad \widetilde{S}\#receiveAgent(\lambda\ \widetilde{AgentCode}, returnid). \\
& \quad (\nu\ id)\widetilde{S}\#\overline{NewAgent}[id, \widetilde{AgentCode}]. \\
& \quad\quad id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
& \quad\quad \widetilde{AgentMethod}\#\overline{onMigration}. \\
& \quad\quad \overline{returnid}[id] \\
& + \quad \widetilde{S}\#killAgent(\lambda\ id).\widetilde{S}\#\overline{DeleteAgent}[id]\Big\}
\end{aligned}
$$

C.3.3.4 Modélisation de l'agent

Comme nous l'avons vu, un agent doit pouvoir communiquer avec le serveur qui l'héberge, pour migrer par exemple. Il doit aussi pouvoir communiquer avec les objets extérieurs via son/leur *id*. Il faut noter que sa méthode *onCreation* est appelée une et une seule fois durant la vie d'un agent, de même que *onDisposing*, tandis que les méthodes $onMigration$, $onMigrating$ ou *migrate* peuvent être appelées à volonté. Heureusement, toutes ces méthodes sont réservées au serveur et nous pouvons donc *a priori* contrôler le nombre de leurs appels.

Nous donnons maintenant le modèle de l'agent puis nous le commentons.

$$
\begin{array}{l}
Agent \equiv (\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S})\Big\{ \\
\quad \Big\{(\nu\ run)\big[\\
\qquad \widetilde{AgentMethod}\#onCreation(\lambda\ arg).(\widetilde{AgentCode}\#C)(arg, id, \widetilde{S}).\overline{run} \\
\qquad + \\
\qquad \widetilde{AgentMethod}\#onMigration.(\widetilde{AgentCode}\#M_o)(\widetilde{S}).\overline{run} \\
\quad \big]\ |run.(\widetilde{AgentCode}\#R)\Big\} \\
\quad |\big[\widetilde{AgentMethod}\#onMigrating(\lambda\ \widetilde{S'}).(\widetilde{AgentCode}\#M_i)(\widetilde{S'}) \\
\qquad + \\
\qquad \widetilde{AgentMethod}\#onDisposing.(\widetilde{AgentCode}\#D)\big] \\
\quad |\big[\widetilde{AgentMethod}\#migrate(\lambda\ \widetilde{S'}).\widetilde{S}\#\overline{sendAgent}[id, \widetilde{S'}] \\
\qquad + \\
\qquad \widetilde{AgentMethod}\#dispose.\widetilde{S}\#\overline{killAgent}[id]\big] \\
\Big\}
\end{array}
$$

La méthode *onCreation* reçoit un argument spécial que nous commenterons en §C.3.4.4 p311, applique l'abstraction C puis lance le processus R en parallèle lorsqu'un signal de terminaison a été reçu sur le nom *run* assurant la séquentialité. R doit être appliquée en parallèle car d'une part, cela correspond à l'allocation d'une thread (rappelons qu'un agent doit être indépendant), et d'autre part, C peut mettre à jour des variables que R souhaite utiliser. Dans la remarque de l'exemple §C.3.1 p302 cela correspond à une communication et nécessite l'emploi de l'opérateur "|". Comme, $onMigration$ doit aussi lancer le processus R en parallèle, il faut utiliser une factorisation.

Évidemment, un appel à $onMigrating$ élimine l'appel à $onDisposing$. La même remarque s'applique à *migrate* et *dispose*. Cependant, ces appels n'éliminent pas R

comme nous allons le voir en §C.3.4.3 p310.

C.3.4 Étude de notre modèle

C.3.4.1 L'état de l'agent

Souvenons-nous qu'un agent est une entité constituée d'un ensemble de threads et de données. A un instant donné, l'agent est dans un *état* particulier : chacune de ses *threads* est dans une primitive particulière et ses données sont fixées. Cet état est reflété par l'abstraction $Extra$. Cette abstraction permet aussi à un agent d'avoir des fonctionnalités inconnues du serveur. Pour pouvoir utiliser ses fonctionnalités, le code de l'agent, à savoir les abstractions C, R, M_o, M_i, D doivent pouvoir communiquer avec $Extra$ (Condition *sine qua non* en π-calcul). C devra donc exécuter $Extra$ en parallèle pour permettre aux autres abstractions de l'utiliser.

C.3.4.2 Contrôle d'un agent

Une fois créé, un agent peut exécuter certaines tâches. Ces tâches ne sont plus connues du serveur : elles sont masquées par le processus R. Par exemple, un déni de service dans notre système d'agents mobiles en Java s'écrit très simplement :

```
while(true);
```

monopolise le processeur. Or, il n'est pas possible d'interrompre un agent en cours d'exécution (la méthode `stop()` est dépréciée, et la méthode `interrupt()` n'est pas *autoritaire*, *cf.* §5.2.3.1 p58).

Si une analyse statique du code de l'agent peut éventuellement aider à détecter ce genre de problème, elle alourdit considérablement la modélisation, et l'exécution d'un serveur d'agents. De plus, les mécanismes de chargement de code dynamique en Java rendent fragile ce genre d'approche. En effet, il est tout à fait possible de coder un agent de telle sorte qu'il embarque dans son état, le bytecode d'une classe malveillante de manière cryptée. Dans ce cas, l'analyse statique ne résout rien même si théoriquement il est impossible "d'opacifier" un programme [26].

Par ailleurs, même les abstractions C, M_o, M_i et D peuvent faire des choses totalement inattendues. Par exemple, on peut avoir $C \equiv P.(R)$. Dans ce cas, le processus (R) sera lancé avant même que C soit terminé. Pire, si (R) est exécuté en parallèle, alors deux processus (R) seront en cours d'exécution au cours de la terminaison de C. Cette problématique apparaît dans les différentes implémentations. C'est le modèle événementiel au niveau de l'agent ($onMigrating$, $onMigration$, ...) rendu nécessaire par l'absence de gestion automatique de l'espace des données (*cf.* §3.2 p30), qui induit ce problème.

Enfin, rien n'empêche un agent d'utiliser ses propres méthodes *onCreation*, *onMigration*, *onMigrating* et *onDisposing*, pourtant réservées au serveur. Le problème vient de la non différenciation entre les méthodes qui peuvent être appelées seulement par le serveur local de l'agent et les autres méthodes, à savoir *migrate*, *dispose* et celles disponibles dans *Extra*.

De même, les autres processus qui ont accès à l'*id* d'un agent peuvent utiliser les méthodes réservées au serveur. Il leur est aussi possible d'utiliser le code de l'agent en exécutant par exemple :

$$id.(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}).\widetilde{AgentCode}\#R$$

Il est toutefois assez simple de résoudre ce problème : il suffit de différencier les méthodes destinées au serveur des autres par des *id* différents. On pourrait avoir par exemple id_{M_s} pour l'accès aux méthodes dédiées au serveur, id_{M_a} pour celles dédiées à l'agent et id_C pour le code de l'agent. Le serveur ne transmettrait alors à l'agent que id_{M_a} et le problème serait réglé. On retrouve cette démarche avec la notion de *proxy* (*cf.* §9.1 p79) dans les aglets [53]. Le proxy est en quelque sorte l'*id* d'une aglet. Un agent ne peut utiliser directement les méthodes d'un autre.

C.3.4.3 Destruction d'un agent

Le π-calcul ne possède pas de mécanisme pour éliminer un terme. Pour utiliser un terme n fois, il doit être répliqué n fois. Dans notre cas, nous ne pouvons savoir combien d'agents auront besoin de communiquer avec un agent donné. Aussi, nous répliquons le terme $\overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}]$ une infinité de fois à l'aide de l'opérateur de réplication "!". Il devient donc impossible de détruire ce terme[7]. Pire, l'abstraction R par exemple, inconnue du serveur peut contenir une ou plusieurs réplications. Il peut donc y avoir accumulation de termes dans l'expression au fur et à mesure des créations d'agents. Ce problème est de fait visible dans les implémentations rencontrées. En Java, c'est le ramasse-miettes qui s'occupe de la gestion mémoire et de la libération des objets lorsqu'ils ne sont plus référencés. Par ailleurs, à part le système des aglets qui utilise la méthode `stop()` pour arrêter une aglet, les autres systèmes ne proposent pas de mécanisme comparable. Étant donné qu'il n'est pas raisonnable d'arrêter un agent de manière autoritaire, on peut imposer que chaque agent soit au minimum capable de répondre à une demande d'arrêt formulée par les mécanismes de communication du système.

[7]Cependant, si un appel via le nom *id* n'est plus possible (il n'y a plus de terme qui ont *id* comme nom), alors le processus d'origine peut être simulé par un processus qui ne contient pas le nom *id* en utilisant une relation de simulation appropriée (opération de ramasse-miettes).

C.3.4.4 Polymorphisme

Le nom de méthode *onCreation* reçoit un argument "utilisateur" et le passe à C :

$$\cdots\ (\widetilde{AgentMethod}\#onCreation(\lambda\ arg).(\widetilde{AgentCode}\#C)(arg, id, \widetilde{S})\ \cdots$$

Cet argument est un peu particulier pour plusieurs raisons :

- il est inconnu du serveur ;
- il peut être une abstraction ;
- il peut être un nom ;
- il peut être un vecteur de noms et d'abstractions.

Il correspond aux paramètres nécessaires à l'instanciation d'un agent. Ces paramètres sont définis par chaque agent et ne peuvent avoir un type précis (une *sorte* en π-calcul). La même remarque s'applique à l'abstraction $Extra$.

Une solution est d'étendre le π-calcul et d'y introduire le polymorphisme comme Pierce et Sangiorgi le proposent [167]. Nous n'avons pas approfondi cette solution mais si dans le formalisme donné, nous n'avons pas introduit la notation polymorphique, c'est pour d'évidentes raisons de clarté.

C.4 Conclusion

Nous avons vu qu'il existait plusieurs types de modélisation de systèmes d'agents mobiles. Si les modélisations structurelles facilitent le développement de plates-formes d'agents mobiles et d'applications à base d'agents mobiles, les modélisations comportementales apportent des fondements théoriques à une technologie qui est assez jeune. Notre proposition à base de π-calcul met en lumière les problèmes inhérents :

- aux systèmes d'agents mobiles qui ne gèrent pas automatiquement l'espace des données et qui fournissent un mécanisme événementiel à cet effet (ce qui peut provoquer l'accumulation de "zombis") ;
- aux systèmes d'agents mobiles écrits en Java qui ne permettent pas de limiter les ressources utilisées par un agent donné.

Enfin, notre modélisation laisse émerger une nouvelle structure de données : le *conteneur actif* qui est présentée dans la partie §III p75 et qui est formalisé en §D p313.

Annexe D

Modélisation des conteneurs actifs

Ce chapitre donne une modélisation en π-calcul du concept de conteneur actif défini au chapitre §8 p75.

Définition D.0.1 (Conteneur actif)
Un conteneur actif *est un conteneur tel que défini en §8.2 p76 qui possède une méthode – que nous appellerons* `call()` *– permettant d'invoquer une méthode d'un des objets qu'il contient. La méthode spécifiée dans l'appel* `call()` *est invoquée de manière* asynchrone.

Remarquons qu'il n'y a plus l'idée de migration, ni même d'agent dans la définition du conteneur actif.

Nous n'allons pas reprendre le même schéma que pour les agents mobiles. En effet, à moins de modéliser complètement un conteneur de type *table de hachage*, nous n'avons pas trouvé de moyen simple d'associer une clé à un élément.

En effet, si nous commencions ainsi, nous aurions défini un conteneur actif par :

$$\widetilde{AC} \asymp \{put, get, call, remove\}$$

on aurait donc formellement en π-calcul :

$$\begin{aligned} &ActiveContainer \equiv (\lambda\ \widetilde{AC}). \\ &\quad (\nu\ Insertion, Deletion, GetObject, CallMethod) \\ &\quad\quad !((Objects)(\widetilde{AC}, Insertion, Deletion, GetObject, CallMethod) \\ &\quad\quad\ |(Services)(\widetilde{AC}, Insertion, Deletion, GetObject, CallMethod)) \end{aligned}$$

Le conteneur proprement dit *Objects* serait défini ainsi :

$$\begin{aligned} Objects \equiv (\lambda\ \widetilde{AC}, Insertion, Deletion, GetObject, CallMethod).\Big\{ & \\ & Insertion(\lambda\ key, F, \widetilde{O}).(F)(\widetilde{O}) \\ & + \\ & GetObject(\lambda\ key, result).\underbrace{?}_{\Uparrow}.\overline{result}[F, \widetilde{O}]\Big\} \end{aligned}$$

Il y a clairement un problème de récupération de l'abstraction F et du vecteur de méthodes $\widetilde{O}$ qui représentent respectivement le code et les méthodes de l'objet à stocker. Ce problème est lié au fait que l'identificateur (la clé *key*) est fournie par l'utilisateur (le client) et non créé par le conteneur. Ce problème va être résolu assez simplement.

En réalité, la clé est un *artifice* de l'implémentation pour permettre au client de désigner son objet. En réalité, nous n'en avons pas besoin. Après une insertion, le client récupère des canaux (noms en π-calcul) privés sur lesquels il peut effectuer les requêtes *get*, *call* et *remove*. Ce sont donc les canaux qui jouent le rôle de clé. Aussi, nous allons utiliser la modélisation suivante :

$$\widetilde{AC} \asymp \{put\}$$

Notre conteneur est réduit à sa plus simple expression : on ne peut utiliser que l'opérateur *put*. En effet, tant qu'un objet n'est pas dans le conteneur, les autres méthodes n'ont pas lieu d'exister.

Le conteneur est défini ainsi :

$$\begin{aligned} & ActiveContainer \equiv (\lambda\ put). \\ & \quad !\Big\{put(\lambda\ F, \widetilde{O}, handles).(F)(\widetilde{O}) \\ & \qquad |(\nu\ get, call, remove)\ \overline{handles}[get, call, remove]. \\ & \qquad\quad \Big[!\overline{get}[F, \widetilde{O}] \\ & \qquad\quad | \ !(call(\lambda\ m, args, result). \\ & \qquad\qquad (\nu\ future)\ \overline{result}[future]. \\ & \qquad\qquad\quad (\nu\ res)(\overline{m}[args, res] \mid res(\lambda\ val).\overline{future}[val]))\Big] \\ & \qquad\quad + remove \\ & \quad \Big\} \end{aligned}$$

Si cela n'apparaît pas dans ce modèle, on retrouve les problèmes liés à la destruction d'objets : après la réception d'un message (vide) sur le canal *remove*, les services

get et *call* ne sont plus accessibles. Néanmoins, un appel précédent à *call* peut avoir créé un certain nombre de processus capables de communiquer directement sans la participation du conteneur actif. Nous avons déjà exploité cette propriété dans le mécanisme nommé *objets stockés multi protocoles* et présenté en §9.11.2 p92.

Remarquons qu'une fois un objet inséré par l'intermédiaire de *put*, on récupère dans le canal *handles* les canaux utilisables pour notre objet : *get*, *call* et *remove*. Notons que l'appel *call* est asynchrone grâce au renvoi immédiat d'un canal $future$ qui sera utilisé lorsque la méthode spécifiée est terminée.

D.1 Simulation d'un système d'agent

Il semble intuitivement possible de simuler notre système d'agents en utilisant le concept de conteneur actif. On considérera le conteneur actif comme la couche bas niveau d'un serveur d'agents. La couche de plus haut niveau offre les services que nous avons modélisés à savoir :

$$\begin{aligned}\widetilde{S} \asymp \{&createAgent,\\ &sendAgent,\\ &receiveAgent,\\ &killAgent\}\end{aligned}$$

Considérons l'ensemble :

$$\begin{aligned}\widetilde{A} \asymp \{&onCreation, C,\\ &onMigration, M_o,\\ &onMigrating, M_i,\\ &onDisposing, D,\\ &run, R\}\end{aligned}$$

Notre serveur d'agents s'écrit donc :

$$
\begin{array}{l}
Server \equiv (\lambda\ \widetilde{S}) \\
\qquad (\nu\ \widetilde{AC})(ActiveContainer)(\widetilde{AC}) \mid (Services)(\widetilde{S}, \widetilde{AC}) \\
\\
Services \equiv (\lambda\ \widetilde{S}, \widetilde{AC})!\ \Big\{ \\
\qquad \widetilde{S}\#createAgent(\lambda\ \widetilde{A}, arg, returnid). \\
\qquad\qquad (\nu\ handles)(\nu\ id)\widetilde{AC}\#\overline{put}[Agent, \widetilde{A}, handles]. \\
\qquad\qquad\qquad handles(\lambda\ get, call, remove). \\
\qquad\qquad\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onCreation, \widetilde{S}, arg, result]. \\
\qquad\qquad\qquad result(\lambda\ future).future(\lambda\ val). \\
\qquad\qquad\qquad (\nu\ dummy)\overline{call}[\widetilde{A}\#run, \mathbf{0}, dummy] \mid \\
\qquad\qquad\qquad \overline{returnid}[id] \mid \\
\qquad\qquad\qquad !\overline{id}[get, call, remove] \\
\qquad + \\
\qquad\qquad \widetilde{S}\#sendAgent(\lambda\ id, \widetilde{S'}, returnNewid). \\
\qquad\qquad\qquad id(\lambda\ get, call, remove). \\
\qquad\qquad\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onMigrating, \widetilde{S'}, result]. \\
\qquad\qquad\qquad result(\lambda\ future).future(\lambda\ val). \\
\qquad\qquad\qquad get(\lambda\ Agent, \widetilde{A}). \\
\qquad\qquad\qquad \overline{remove}. \\
\qquad\qquad\qquad \widetilde{S}\#receiveAgent(\widetilde{A}, returnNewid) \\
\qquad + \\
\qquad\qquad \widetilde{S}\#receiveAgent(\lambda\ \widetilde{A}, returnid). \\
\qquad\qquad\qquad (\nu\ handles)(\nu\ id)\widetilde{AC}\#\overline{put}[Agent, \widetilde{A}, handles]. \\
\qquad\qquad\qquad\qquad handles(\lambda\ get, call, remove). \\
\qquad\qquad\qquad\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onMigration, \widetilde{S}, result]. \\
\qquad\qquad\qquad\qquad result(\lambda\ future).future(\lambda\ val). \\
\qquad\qquad\qquad\qquad (\nu\ dummy)\overline{call}[\widetilde{A}\#run, \mathbf{0}, dummy] \mid \\
\qquad\qquad\qquad\qquad \overline{returnid}[id] \mid \\
\qquad\qquad\qquad\qquad !\overline{id}[get, call, remove] \\
\qquad + \\
\qquad\qquad \widetilde{S}\#killAgent(\lambda\ id). \\
\qquad\qquad\qquad id(\lambda\ get, call, remove). \\
\qquad\qquad\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onDisposing, \widetilde{S}, result]. \\
\qquad\qquad\qquad result(\lambda\ future).future(\lambda\ val). \\
\qquad\qquad\qquad \overline{remove} \\
\Big\}
\end{array}
$$

Remarquons l'utilisation du nom *handles* pour récupérer les noms *get*, *call* et *remove* associés à notre objet. Remarquons aussi l'expression :

$$result(\lambda\ future).future(\lambda\ val)$$

qui permet d'attendre la fin de l'appel asynchrone *call*. Enfin, après la création d'un agent, c'est-à-dire l'insertion dans le conteneur de l'abstraction *Agent* et des noms $\widetilde{A}$, l'identificateur de l'agent nous sert à récupérer par la suite les noms *get*, *call* et *remove*. L'envoi d'un agent se fait en récupérant ces noms, en invoquant la méthode $onMigrating$ de manière synchrone, en récupérant une copie de l'agent, en le retirant du conteneur et en l'envoyant sur le serveur distant. La réception ressemble à la création.

L'agent lui s'écrit sans surprise :

$$
\begin{array}{l}
Agent \equiv (\lambda\ id, \widetilde{A}). \\
\quad (1) \qquad (\nu\ getCurrentServer) \left\{ \right. \\
\quad (2) \qquad (\nu\ setCurrentServer) \Big[\ \big[\ setCurrentServer(\lambda\ \widetilde{S}). \\
\quad (3) \qquad\qquad\qquad \overline{getCurrentServer}[\widetilde{S}]\ \big]\ \mid \\
\qquad\quad \Big[\widetilde{A}\#onCreation(\lambda\ \widetilde{S}, arg).\overline{setCurrentServer}[\widetilde{S}].(\widetilde{A}\#C)(arg)\ + \\
\qquad\quad \widetilde{A}\#onMigration(\lambda\ \overline{S}).\overline{setCurrentServer}[\widetilde{S}].(\widetilde{A}\#M_o)(\widetilde{S})\Big]\ \Big]\ \mid \\
\qquad\quad \widetilde{A}\#run.(\widetilde{A}\#R)\ \mid \\
\qquad\quad \Big[\widetilde{A}\#onMigrating(\lambda\ \widetilde{S}).(\widetilde{A}\#M_i)(\widetilde{S})\ + \\
\qquad\quad \widetilde{A}\#onDisposing.\overline{setCurrentServer}[\mathbf{0}].(\widetilde{A}\#D)\ \Big]\ \mid \\
\qquad\quad \Big[\widetilde{A}\#migrate(\lambda\ \widetilde{S}').getCurrentServer(\lambda\ \widetilde{S}).\widetilde{S}\#\overline{send}[id, \widetilde{S}']\ + \\
\qquad\quad \widetilde{A}\#dispose.getCurrentServer(\lambda\ \widetilde{S}).\widetilde{S}\#\overline{kill}[id]\ \Big] \\
\left. \right\}
\end{array}
$$

Les lignes notés (1), (2) et (3) modélisent une variable dont l'accès ne peut être effectué qu'une seule fois après avoir été préalablement intialisé. Les méthodes *setCurrentServer* et *getCurrentServer* permettent donc respectivement de positionner le serveur courant de l'agent et de le récupérer. Cette dernière permet dans la méthode *migrate* de demander au serveur courant de nous migrer.

Intuitivement, nous pensons qu'il est possible de simuler les systèmes d'agents mobiles avec le modèle de conteneurs actifs. Il reste donc à vérifier qu'il existe bien une bisimulation entre cette modélisation et celle donnée dans le chapitre consacré à la modélisation des agents mobiles (*cf.* §C.3 p297). Pour cela, il faudrait prendre en compte les *sortes* des différents noms pour avoir une preuve correcte. Celle-ci serait peut-être facilitée par les travaux de l'équipe de Tom Melham [134] qui proposent une transcription de la théorie du π-calcul dans la logique d'ordre supérieur en utilisant le système de preuve HOL.

Bibliographie

[1] Foxtrot – Easy API for JFC/Swing.
http://foxtrot.sourceforge.net/.

[2] Spin your Swing.
http://spin.sourceforge.net/.

[3] Pi net site. Web page, April 2002.
http://www.cecm.sfu.ca/pi/pi.html.

[4] AgentX, Octobre 2003.
http://www.mobileagenttech.com/prod01.htm.

[5] Cluster for intelligent mobile agents for telecommunication environments (CLIMATE), 2003.
http://www.fokus.fhg.de/research/cc/ecco/climate/-intro-climate-cluster.html.

[6] Foundation for Intelligent Physical Agents (FIPA), 2003.
http://www.fipa.org/.

[7] Kaffe, Octobre 2003.
http://www.kaffe.org.

[8] Cristiano Sadun – utility classes. Web page, August 2004.
http://sadun-util.sourceforge.net/.

[9] GNU Classpath. Web page, July 2004.
http://www.gnu.org/software/classpath/classpath.html.

[10] Grid 5000 project. Web page, July 2004.
www.grid5000.org.

[11] JSR 121: Application Isolation API Specification. Web page, July 2004.
http://www.jcp.org/en/jsr/detail?id=121.

[12] Prefixes for binary multiples. Web page, August 2004.
http://physics.nist.gov/cuu/Units/binary.html.

[13] Ruby. Web page, July 2004.
http://www.ruby-lang.org/en/.

[14] The JavaGrande HomePage, June 2004.
`http://www.javagrande.org/`.

[15] The Agent Society. *Design Workshop on Open Intelligent Agent Platforms and Protocols*, February 1997. First Meeting of the Agent Interop Working Group.

[16] Gul Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, University of Michigan, 1986.

[17] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 398–407. Kaufmann, San Mateo, CA, 1988.

[18] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[19] Yariv Aridor and Danny B. Lange. Agent design patterns: elements of agent application design. In *Proceedings of the second international conference on Autonomous agents*, pages 108–115. ACM Press, 1998.

[20] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, third edition, 2000. ISBN : 0-201-70433-1.

[21] Ken Arnold, Gosling James, and Holmes David. *Le langage Java*. Vuibert informatique, 3eme edition, 2001. BN 02669994 01-39490; traduction de Serge Chaumette, Alexis Moussine-Pouchkine, Pascal Grange, Asier Ugarte et Pierre Vignéras.

[22] Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of Java programs. In *FMOODS*, pages pp. 141–161, Stanford University, 6-8 2000. Kluwer Academic Publishers.

[23] Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, flexible and typed group communications for Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, November 3-5 2002.
`http://www.inria.fr/oasis/caromel/ps/GroupCommunicationsfor-Java.pdf`.

[24] David Bailey, Peter Borwein, and Simon Plouffe. On the rapid computation of various polylogarithmic constants. Web page, April 2002.
`http://www.lacim.uqam.ca/plouffe/articles/-BaileyBorweinPlouffe.pdf`.

[25] Saisanthosh Balakrishnan and Karthik Pattabiraman. Migration of threads containing pointers in distributed systems. Web, 2003. `http://www.cs.wisc.edu/~saisanth/papers/hipc00.pdf`.

[26] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:1–??, 2001.

[27] Donald Becker and Phil Merkey. Beowulf project, May 2000. `http://www.beowulf.org/`.

[28] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, USA, 18-21 May 2003. `http://www.usenix.org/events/hotos03/tech/full_papers/-vonbehren/vonbehren_html/`.

[29] Mordechai Ben-Ari and Yifat Ben-David Kolikant. Thinking parallel: the process of learning concurrency. In *4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 13–16, Cracow, Poland, 1999. ACM Press. ISSN:0097-8418 `http://stwi.weizmann.ac.il/g-cs/benari/articles/thnkpar.pdf`.

[30] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[31] Patrick Biget. The vault, an architecture for smartcards to gain infinite memory. In *CARDIS*, pages 305–312, 1998.

[32] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[33] Matt Blaze. High-bandwith encryption with low-bandwith smartcards, 1995. `ftp://ftp.research.att.com/dist/mab/card_cipher.ps`.

[34] N. Borselius. Mobile agent security. *Electronics & Communication Engineering Journal*, 14(5):211–218, October 2002.

[35] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[36] Joseph Bowbeer. The Last Word in Swing Threads – Working with Asynchronous Models. Web page. `http://java.sun.com/products/jfc/tsc/articles/threads/-threads3.html`.

[37] C. Bumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper - A universal agent platform based on OMG Masif and FIPA standards.

[38] Jean-Louis Bénard, Laurent Bossavit, Régis Médina, and Dominic Williams. *L'Extreme Programming - Avec deux études de cas*. Technologies objet. Eyrolles, 1 ère edition, mai 2002. ISBN : 2-212-11051-0, EAN13 : 9782212110517.

[39] Canadian Mind Products. The Four Types Of Execution. Web page, June 2004.
`http://mindprod.com/jgloss/compiler.html`.

[40] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998)*, volume 1378 of *lncs*, pages 140–155. sv, 1998.

[41] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. In Geoffrey C. Fox, editor, *Concurrency: practice and experience*, volume 10, pages 1043–1061. Wiley and Sons, Ltd., September-November 1998.

[42] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[43] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, 1997.

[44] Maui High Performance Computing Center. Sp parallel programming workshop. Web page, July 2004.
`http://www.mhpcc.edu/training/workshop/ibmhwsw/MAIN.html`.

[45] Pittsburgh Supercomputing Center. The cray t3e. Web page, July 2004.
`http://www.psc.edu/machines/cray/t3e/t3e.html`.

[46] S. Chaumette. JEM-DOOS: the Java based Distributed Objects Operating System of the JEM project. In Denis Caromel and al., editors, *Proceedings of the Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98, Santa Fe, NM, USA*, volume 1505, pages 135–142, decembre 1998.

[47] Serge Chaumette and Pascal Grange. Parallelizing multithreaded java programs: a criterion and its pi-calculus foundation. In *Workshop on Formal Methods for Parallel Programming/IPDPS*, 2002.

[48] Serge Chaumette and Pascal Grange. Optimizing the execution of a distributed object oriented application by combining static and dynamic information. International Parallel and Distributed Processing Symposium, IPDPS 03, April 2003. Nice, France. Poster.

[49] Serge Chaumette, Pascal Grange, Damien Sauveron, and Pierre Vignéras. Computing with Java cards. In *International Conference on Computer, Communication and Control Technologies/CCCT'2003*, July, 31th - August, 2nd 2003.

[50] Serge Chaumette, Benoît Métrot, Pascal Grange, and Pierre Vignéras. JToe: a Java API for object exchange. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Advances in Parallel Computing*, Elsevier, Netherlands, 2003. Parallel Computing: Software Technology, Algorithms, Architectures and Applications.

[51] Serge Chaumette and Pierre Vignéras. Extensible and customizable just-in-time-security (JITS) management of client-server communication in Java. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Advances in Parallel Computing*, Elsevier, Netherlands, 2003. Parallel Computing: Software Technology, Algorithms, Architectures and Applications. `http://rparco.urz.tu-dresden.de/Parco2003/up/-1p-abstract-111.pdf`.

[52] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.

[53] IBM Corporation. Aglets home page, January 2001. `http://www.trl.ibm.co.jp/aglets/` et aussi `http://aglets.sourceforge.net/`.

[54] IBM Corporation. Web page, July 2004. `www.ibm.com`.

[55] Intel Corporation. Web page, July 2004. `http://www.intel.com/`.

[56] Microsoft Corporation. Smartcard for windows, Octobre 2003. `http://www.microsoft.com/technet/treeview/-default.asp?url=/technet/security/topics/smrtcard/smrtcdcb/-sec1/smartc01.asp`.

[57] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analysing mobile code languages. In *Second International ECOOP Workshop on Mobile Object Systems*, Linz, Austria, July 1996.

[58] Horvat Damir, Cvetkovic Dragana, Milutinovic Veljko, Kocovic Petar, and Kovacevic Vlada. Mobile agents and Java mobile agents toolkits. In *33rd Hawaii International Conference on System Sciences*, volume 8, page 8029, Maui, Hawaii, January 2000. IEEE.

[59] Lange B. Danny and Oshima Mitsuru. *Programming and Deploying Mobile Agents with Java*, chapter Mobile Agents With Java: The Aglets API. 1998.

[60] Lange B. Danny and Oshima Mitsuru. *Programming and Deploying Mobile Agents with Java*, chapter Mobile Objects and Mobile Agents: The Future of Distributed Computing. 1998.

[61] Université Technique de Lisbone. Agentspace, Septembre 2003. `http://berlin.inesc.pt/agentspace/`.

[62] Equipe Composants du laboratoire Valoria. SAJE – System-Aware Java Environment, 2004. `http://www-valoria.univ-ubs.fr/Composants/CASA/SAJE/-saje-fr.shtml`.

[63] U. Engberg and M. Nielsen. A calculus of communicating system with label-passing. Technical report, Computer Science Department, University of Aarhus, Denmark, 1996.

[64] M. Abadi et A. Gordon. A calculus for cryptographic protocols - the spi calculus. Page Web, 11 décembre 1996.

[65] Katrina E. Kerry Falkner, Paul D. Coddington, and Michael J. Oudshoorn. Implementing Asynchronous Remote Method Invocation in Java. Technical report, Distributed High Performance Computing Group, July 1999.

[66] Claudio Fleiner, Jerry Feldman, and David Stoutamire. Killing threads considered dangerous. In *Conference on Parallel and Object Oriented Methods and Applications (POOMA)*, Santa Fe, 1996.

[67] Center for Advanced Computing Research. Intel Paragon information page. Web page, July 2004. `http://www.cacr.caltech.edu/resources/paragon/`.

[68] International Organisation for Standardisation. "identification cards – integrated circuit(s) cards with contacts – part 7: Interindustry commands for structured card query language (SCQL)", 1999. "ISO/IEC 7816-7".

[69] Free Software Foundation. GNU Lesser General Public License, February 1999. `http://www.fsf.org/licenses/lgpl.html`.

[70] Free Software Foundation. GNU Emacs. Web page, July 2004. `http://www.gnu.org/software/emacs/`.

[71] Free Software Foundation. GNU Make. Web page, July 2004. `http://www.gnu.org/software/make/`.

[72] The Apache Software Foundation. The Apache Ant Project. Web page, July 2004. `http://ant.apache.org/`.

[73] The Eclipse Foundation. Eclipse. Web page, July 2004. `http://www.eclipse.org/`.

[74] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[75] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.

[76] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[77] Shah G., Nieplocha J., Mirza C., Harrison R., Govindaraju R. K., Gildea K., DiNicola P., and Bender C. Performance and experience with LAPI: a new high-performance communication library for the IBM RS/6000 SP. In *International Parallel Processing Symposium*, pages 260–266, 1998.

[78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, chapter Bridge – Object Structural, pages 151–161. In [82], 1994. ISBN : 0-201-63361-2.

[79] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, chapter Adapter – Object Structural, pages 139–149. In [82], 1994. ISBN : 0-201-63361-2.

[80] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, chapter Singleton – Object Creational, pages 127–134. In [82], 1994. ISBN : 0-201-63361-2.

[81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, chapter Factory – Object Creational, pages 87–95. In [82], 1994. ISBN : 0-201-63361-2.

[82] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN : 0-201-63361-2.

[83] Erich Gamma and Kent Beck. The JUnit Home Page, Juillet 2004. `http://www.junit.org/`.

[84] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine : A Users'Guide and Tutorial for Networked*

Parallel Computing. Scientific and Engineering Computation Series. The MIT Press, third edition, 1996. ISBN : 0-262-57108-0.

[85] General Magic. Odyssey home page, June 1998. `http://www.genmagic.com/technology/`.

[86] GMD FOKUS and IBM Corporation. *Mobile Agent System Interoperability Facilities Specification (MASIF)*, November 1997. Supported by Crystaliz, Inc.; General Magic, Inc.; The Open Group.

[87] GMD FOKUS, IBM Corporation, and The Open Group. *Mobile Agent Facility Specification (MAF)*, December 1996.

[88] J. Gosling, B. Joy, and G. Steele. *Chapter 17 – Java Memory Model*. In [89], second edition, 2000.

[89] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[90] Pascal Grange and Pierre Vignéras. JToe. Web page, September 2003. `http://jtoe.sf.net`.

[91] Gilles Grimaud and Sebastien Jean. Carte et mobilité. In *4ème école d'informatique des systèmes parallèles et répartis (ISYPAR'2000)*, 2000.

[92] Gilles Grimaud and Sébastien Jean. "GUM-E^2: une approche orientée carte à microprocesseur pour la gestion sécurisée de la mobilité de l'utilisateur dans les échanges électroniques". In *Trusded Electronic Tread (TET'99)*, pages 415–429, Marseille, France, June 1999.

[93] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, second edition, 1999. ISBN : 0-262-57132-3.

[94] Eric Gunnerson. Why doesn't C# require exception specifications? .NET Forum Web page, Juin 2004. `http://discuss.develop.com/archives/wa.exe?A2=ind0011A&L=DOT-NET&P=R32820`.

[95] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical report, T. J. Watson Research Center, Yorktown Heights, New York, 1995.

[96] Bernhard Haumacher, Thomas Moschny, Jurgen Reuter, and Walter Tichy. Transparent distributed threads for Java. In *17th International Parallel and Distributed Processing Symposium*, page 136, Nice, France, April 2003. IEEE.

[97] Carl Hewitt. Viewing control structures as patterns of passing messages. In *Journal of Artificial Intelligence*, volume 8, pages p. 323–364, June 1977.

[98] S. Hirano. HORB: Distributed Execution of Java Programs. In *Proc. WWCA'97*, volume Vol. 1274 of *LNCS*, pages 29–42. Springer Verlag, Berlin, 1997. `http://ring.etl.go.jp/openlab/horb/`.

[99] Allen Holub. If i were king: A proposal for fixing the Java programming language's threading problems. Web page, October 2000. `ftp://www6.software.ibm.com/software/developer/library/-j-king.pdf`.

[100] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–??, 1991.

[101] IBM. IBM – Developer Works: Java technology. Web page, July 2004. `http://www-136.ibm.com/developerworks/java/`.

[102] IBM. Jikes RVM. Web page, July 2004. `http://www-124.ibm.com/developerworks/oss/jikesrvm/`.

[103] Adobe Systems Inc. *Postscript Language Reference Manual*, addison-wesley edition, 1985.

[104] Cray Inc. Web page, July 2004. `http://www.cray.com/`.

[105] Naomaru Itoi. SC-CFS: Smartcard secured cryptographic file system. pages 271–280.

[106] Ian Kaplan. Web Pages Related to Compiling the Java Programming Language. Web page, June 2004. `http://www.bearcave.com/software/java/comp_java.html/`.

[107] G. Karjoth, D. B. Lange, and M. Oshima. A security model for Aglets. *IEEE Internet Computing*, 1(4):68–77, 1997.

[108] Miguel Katrib, Iskander Sierra, Mario del Valle, and Thaizel Fuentes. Java distributed separate objects. *Journal of Object Technology*, vol. 1(No. 2):pages 119–142, July-August 2002. `http://www.jot.fm/issues/issue_2002_07/article2`.

[109] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The art of metaobject protocol*. MIT Press, 1991.

[110] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Videira Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Springer-Verlad, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlad, 1997.

[111] Joseph Roland Kiniry. Exceptions in Java and Eiffel: Two extremes in exception design and application. In *ECOOP 2003: Exception Handling in Object*

Oriented Systems: towards Emerging Application Areas and New Programming Paradigms, July 21 2003.
`http://kind.cs.kun.nl/~kiniry/papers/exceptions.pdf`.

[112] Paul Kinnucan. Java Development Environment for Emacs. Web page, July 2004.
`http://jdee.sunsite.dk/`.

[113] Frederick Knabe. Language Support for Mobile Agents. Technical Report CMU-CS-95-223, Pittsburgh, PA 15213, 1995.

[114] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Principles of Programming Languages*, 1996.

[115] Nikhil Kothari. AgentOS - A Java based mobile agent system. ICS Honors Project Final Report.

[116] David Kotz and Robert S. Gray. Mobile agents and the future of the internet. *Operating Systems Review*, 33(3):7–13, 1999.

[117] Vijaykumar Krishnaswamyg, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java Remote Method Invocation. In *4th USENIX Conference on ObjectOriented Technologies and Systems (COOTS'98)*, 1998.

[118] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[119] Hugh C. Lauer and Roger M. Needham. On the duality of operating systems structures. In *Second International Symposium on Operating Systems*, volume IR1A, Octobre 1978. reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

[120] P. Launay and J. Pazat. A Framework for parallel programming in Java. Publication interne 1154, Institut de Recherche en Informatique et Systèmes Aléatoires, décembre 1997. Thème I - Réseaux et systèmes.

[121] Pascal Launay and Jean-Louis Pazat. Generation of distributed parallel Java programs. Internal publication 1171, Institut de Recherche en Informatique et Systèmes Aléatoires, February 1998. Thème I - Réseaux et systèmes.

[122] Nicolas Le Sommer. *Contractualisation des ressources pour les composants logiciels : une approche réflexive.* PhD thesis, Université de Bretagne-Sud, December 2003.

[123] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*, chapter 1.3 Design Forces, pages 37–55. In [124], 1999.

[124] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[125] Xavier Leroy. The LinuxThreads Library. Web page, June 2004. `http://pauillac.inria.fr/~xleroy/linuxthreads/`.

[126] Anselm Lingnau and Oswald Drobnik. An Infrastructure for Mobile Agents: Requirements and Architecture. In *Proceedings of the 13th DIS Workshop*, Orlando, FL, USA, 1995.

[127] Johannes Link. *Tests unitaires en Java – Les tests au coeur du développement.* Dunod, première edition, Septembre 2003. ISBN : 2-10-008152-7, EAN13 : 9782100081523.

[128] Johannes Link. *Tests unitaires en Java – Les tests au coeur du développement*, chapter 10. Programmes Parallèles, pages 183–202. In Dunod [127], première edition, Septembre 2003. ISBN : 2-10-008152-7, EAN13 : 9782100081523.

[129] Douglas Lyon. CentiJ: An RMI Code Generator. *Journal of Object Technology*, 1(5):117–148, November-December 2002. `http://www.jot.fm/issues/issue_2002_11/article2`.

[130] Farmer William M., Guttman Joshua D., and Swarup Vipin. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, 1996.

[131] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1999.

[132] J. E. White General Magic. Telescript technology: Mobile agents, 1995.

[133] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In *Advances in Distributed Systems.* IEEE, 1992.

[134] T. F. Melham. A mechanized theory of the *pi*-calculus in HOL. Technical report, Departement d'informatique 'a l' universit'e de Glasgow, Ecosse, 1992.

[135] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Highlights from TOP500 list for june 2004. Web page, July 2004. `http://www.top500.org/lists/2004/06/trends.php`.

[136] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites. Web page, July 2004. `http://www.top500.org/`.

[137] Betrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):56–80, 1993.

[138] Microsoft. Dcom. Web page, July 2004. `http://www.microsoft.com/com/tech/DCOM.asp`.

[139] Microsoft. From the C# team. Web page, July 2004.
`http://msdn.microsoft.com/vcsharp/team/default.aspx`.

[140] Microsoft. Microsoft and Sun microsystems enter broad cooperation agreement; settle outstanding litigation. Web page, July 2004.
`http://www.microsoft.com/presspass/press/2004/apr04/-04-02SunAgreementPR.asp`.

[141] Microsoft. Microsoft Java virtual machine support. Web page, July 2004.
`http://www.microsoft.com/mscorp/java/`.

[142] Sun microsystem. Why are `Thread.stop()`, `Thread.suspend()`, `Thread.resume()` and `Runtime.runFinalizersOnExit()` deprecated? Web.
`http://java.sun.com/products/jdk/1.2/docs/guide/misc/-threadPrimitiveDeprecation.html`.

[143] Sun microsystem. Javacard, Octobre 2003.
`http://http://java.sun.com/products/javacard`.

[144] Sun microsystems. Java Naming and Directory Interface – JNDI Documentation.
`http://java.sun.com/products/jndi/docs.html`.

[145] Sun microsystems. Using the SwingWorker Class. Web page.
`http://java.sun.com/docs/books/tutorial/uiswing/misc/-threads.html#SwingWorker`.

[146] Sun microsystems. Using dynamic proxies to layer new functionality over existing code. Web page, may 2000.
`http://java.sun.com/developer/TechTips/2000/tt0530.html`.

[147] Sun microsystems. Java Message Service. Web page, Juin 2004.
`http://java.sun.com/products/jms/`.

[148] Sun microsystems. Reflection guide, 2004.
`http://sunsite.nstu.ru/java-stuff/JDK/guide/reflection/`.

[149] R. Milner. *Lectures on a Calculus for Communicating Systems*, volume 197 of *LNCS*. Springer-Verlag, New York, NY, 1984.

[150] Robin Milner. The polyadic π-calculus : a tutorial. Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, octobre 1991.

[151] Mitsubishi Electric. Concordia home page, January 2001.
`http://www.meitca.com/HSL/Projects/Concordia/`.

[152] The mole team. The mobile agent list. Web, 1999.
`http://mole.informatik.uni-stuttgart.de/`.

[153] Pablo Moscato. TSPBIB – Traveling Salesman Problem Bibliography. Web page, June 2004.
`http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html`.

[154] Myricom. Myrinet link and routine specification, 1995.
`http://www.myri.com/myricom/document.html`.

[155] R. Namyst. *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Université de Lille 1, Janvier 1997.
`http://www.ens-lyon.fr/~bouge/Biblio/ParaDigme.html`.

[156] Christian Nester, Michael Pilippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of Java Grande Conference*, pages 152–157, San Francisco, California, June 1999. ACM.

[157] Uwe Nestmann. Links on calculi for mobile processes, January 2001.
`http://lampwww.epfl.ch/mobility/`.

[158] Object Management Group. Grasshopper home page, octobre 2003.
`http://www.grasshopper.de/`.

[159] Object Management Group. *OMG Unified Modeling Language (UML) Specification version 1.5*, march 2003.
`http://www.omg.org/docs/formal/03-03-01.pdf`.

[160] Object Management Group (OMG). *Naming Service Specification (COS v1.2)*, September 2002.
`http://www.omg.org/docs/formal/02-09-02.pdf`.

[161] National Bureau of Standards. "data encryption standard". Federal Information Processing Standards Publication FIPS PUB 46, 1977.

[162] R. Orfali, D. Karkey, and J. Edwards. *Objets répartis - guide de survie*. Thomson Publishing, 1996. ISBN : 2-84180-002-4.

[163] Alessandro Orso, Mary Jean Harrold, and Giovannia Vigna. Mobile agents security through static/dynamic analysis.

[164] John Ousterhout. Why threads are a bad idea (for most purposes). Présentation donnée à la conférence technique annuelle Usenix, Janvier 1996.
`http://www.cs.utah.edu/~regehr/research/ouster.pdf`.

[165] Michael Philippsen and Bernard Haumacher. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in LNCS, pages 718–732, San Juan, Puerto Rico, April 1999. International Workshop on Java for Parallel and Distributed Computing.

[166] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science*, Vol. 6, No. 5, 1996.

[167] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Principles of Programming Languages (POPL)*, 1997. Full version available as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 468.

[168] Benjamin C. Pierce and David N. Turner. The Pict programming language. Web page, July 2004.
`http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html`.

[169] M. Pilippsen and M. Zenger. JavaParty - Transparent Remote Objects In Java. In *Concurrency: practice and experience*, volume 9, pages 1225–1242, 1997.

[170] Jon Postel. "RFC 768: User datagram protocol", August 1980. http://www.faqs.org/rfcs/rfc768.html.

[171] Jon Postel. "RFC 791: Internet protocol", September 1981. http://www.faqs.org/rfcs/rfc791.html.

[172] Jon Postel. "RFC 793: Transmission control protocol", September 1981. http://www.faqs.org/rfcs/rfc793.html.

[173] Derek Price. Concurrent Version System (CVS). Web page, July 2004.
`https://www.cvshome.org/`.

[174] Loic Prylli and Bernard Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. In *IPPS/SPDP Workshops*, pages 472–485, 1998.

[175] Recursion Software (purchased from ObjectSpace). Voyager home page, Octobre 2003.
`http://www.recursionsw.com/products/voyager/`.

[176] B. Garbinato R. Guerraoui and K. Mazouni. Distributed Programming in Garf. In Springer Verlag, editor, *LNCS*, volume 791 of *Object Based Distributed Programming*, pages 225–239, 1994.

[177] RedHat. The Native POSIX Thread Library for Linux. PDF documentation file, June 2004.
`http://www.redhat.com/whitepapers/developer/-POSIX_Linux_Threading.pdf`.

[178] David Walker Robin Milner, Joachim Parrow. A calculus of mobile processes (parts I and II). Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, juin 1989.

[179] Volker Roth. Obstacles to the adoption of mobile agents. In *IEEE International Conference on Mobile Data Management (MDM'04)*, page p. 296, Berkeley, California, January 19 - 22 2004. Institute of Electrical and Electronics Engineers (IEEE).
`http://csdl.computer.org/comp/proceedings/mdm/2004/2070/00/-20700296.pdf`.

[180] Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. Technical Report RR-2539, INRIA-Sophia Antipolis, 1995.
`http://www.inria.fr/rrrt/rr-2539.html`.

[181] Damien Sauveron. La technologie Java card : Présentation de la carte à puce, la java card. Technical report, 2001.

[182] Douglas Schmidt and Steve Vinoski. Programming asynchronous method invocation with CORBA messaging.

[183] Marco Schmidt. List of Java just-in-time (JIT) compilers. Web page, June 2004.
`http://www.geocities.com/marcoschmidt.geo/-java-jit-compilers.html`.

[184] Damien Sauveron (SERMA/LaBRI) Serge Chaumette (LaBRI), Jean-Pierre Lacoustille (SERMA). Projet sécurité Java card, rapport d'avancement, phase 1, juin 2001 à octobre 2001. Technical report, 2001.

[185] Calvin Shen and Larry T. Chen., 1998. "UCI Undergraduate Research Journal".

[186] Jon Siegel. *CORBA, Fundamental and Programming*. Wiley, 1996.

[187] Alberto Silva and Jose Delgado. The agent pattern for mobile agent systems. In *European Conference on Pattern Languages of Programming and Computing, EuroPLoP*, 1998.

[188] Jonathan Simon. Rethinking Swing Threading. Web page, October 2003.
`http://today.java.net/pub/a/today/2003/10/24/swing.html?page=1`.

[189] André Spiegel. *Automatic Distribution of Object-Oriented Programs*, chapter 2.3.1 Distributed Computing: A Note on A Note, pages 50–53. In [190], December 2002.
`http://page.mi.fu-berlin.de/~spiegel/papers/phd.pdf`.

[190] André Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, FB Mathematik und Informatik, December 2002.
`http://page.mi.fu-berlin.de/~spiegel/papers/phd.pdf`.

[191] SUN Microsystems. *Java Remote Method Invocation Specification*, 1998.
`http://java.sun.com/products/jdk/1.1/docs/guide/rmi/`.

[192] SUN Microsystems. Jini Architectural Overview. Technical report, Sun Microsystems, January 1999.
http://www.sun.com/jini/.

[193] Sun microsystems. The Java HotSpot performance engine architecture. white paper, April 1999.
http://java.sun.com/products/hotspot/whitepaper.html.

[194] SUN Microsystems. *Enterprise JavaBeans Specification*, juin 2000. Version 2.0, Public Draft.

[195] Sun microsystems. Java 2 Enterprise Edition Web site. Web page, June 2002.
http://java.sun.com/products/j2ee/.

[196] Sun microsystems. Java 2 Micro Edition Web site. Web page, June 2002.
http://java.sun.com/products/j2me/.

[197] Sun microsystems. Java 2 Platform, Standard Edition, v 1.4.0 API Specification Web site. Web page, June 2002.
http://java.sun.com/j2se/1.4/docs/api/index.html.

[198] Sun microsystems. Java 2 Standard Edition Web site. Web page, June 2002.
http://java.sun.com/products/j2se/.

[199] GCC Team. The GNU Compiler for the Java Programming Language. Web page, June 2004.
http://gcc.gnu.org/java/.

[200] T.F. Melham. *Introduction to the HOL theorem prover*. University of Cambridge, Computer Laboratory, Cambridge, England, 1990.

[201] Cynthia Tham, Barry Friedman, and Jim White. MAF issues. Technical report, General Magic Inc., November 1996.

[202] Cynthia Tham, Barry Friedman, Jim White, and Tony Rutkowski. An assessment of the mobile agent facility proposal. Technical report, General Magic Inc., January 1997.

[203] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.

[204] B. Thomsen. *Calculi for higher-order communicating systems*. PhD thesis, Imperial College, London University, 1990.

[205] Bent Thomsen. *Calculi for Higher Order Communicating Systems*. Ph.D. thesis, Imperial College, London, 1990.

[206] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency*

Theory, 7th International Conference, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[207] S. Tschoeke, R. Luling, and B. Monien. Solving the traveling salesman problem with a distributed branch and bound algorithm on a 1024 processor network. In *International Parallel Processing Symposium*, pages pages 182–189, 1995. `http://citeseer.ist.psu.edu/148117.html`.

[208] A. Ugarte. Mise en œuvre d'un environnement multitâche, multi-utilisateur et distribué en Java. *CPRSR, Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 12(5-6):511–535, décembre 2000.

[209] A. Ugarte. *Mise en œuvre d'un environnement objet distribué étayée par une modélisation des* threads *Java*. PhD thesis, Université Bordeaux I, LaBRI, 4 Janvier 2001. Rapporteurs : Jean-François Méhaut et Yves Robert. Jury : Richard Castanet, Serge Chaumette, Frédéric Desprez, Alain Griffault, Jean-François Méhaut et Alain Miniussi.

[210] Carlos Varela and Gul Agha. What after Java? from objects to actors. In *Proceedings of the seventh international conference on World Wide Web*, pages p. 573–577, Brisbane, Australia, 1998. Elsevier Science Publishers B. V. Amsterdam, The Netherlands, The Netherlands. ISSN:0169-7552 `http://www7.scu.edu.au/programme/docpapers/1890/com1890.htm`.

[211] Pierre Vigneras. DJFractal. Web page, August 2004. `http://djfractal.sf.net/`.

[212] Pierre Vigneras. Mandala. Web page, August 2004. `http://mandala.sf.net/`.

[213] Pierre Vigneras. Mandala – CVS. Web page, August 2004. `http://sourceforge.net/cvs/?group_id=64217`.

[214] Pierre Vigneras. Mandala – Distribution. Web page, August 2004. `http://sourceforge.net/project/showfiles.php?group_id=64217`.

[215] Pierre Vigneras. Mandala – SourceForge Project Page. Web page, August 2004. `http://sourceforge.net/projects/mandala`.

[216] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. *Lecture Notes in Computer Science*, 1445:330–??, 1998.

[217] S. Vinoski. CORBA : Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Comunications Magazine*, 35(2), February 1997. `http://www.cs.wustl.edu/~schmidt/corba-papers.html`.

[218] Steve Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10):44–52, 1998.

[219] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[220] M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (LDAP v3)*. The Internet Society – Network Working Group, December 1997. Request for Comments: 2251, Category: Standards Track `ftp://ftp.isi.edu/in-notes/rfc2251.txt`.

[221] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[222] E Walker, R Floyd, and P Neves. Asynchronous Remote Operation Execution In Distributed Systems. In *International Conference on Distributed Computing Systems*, number 10, pages 253–259, Paris, France, May/June 1990.

[223] Jim White. IBM tokyo labs-General Magic meeting. Sunnyvale CA, November 1996.

[224] Reisig Wolfgang. *Petri Nets: an introduction*. Springer-Verlag, Berlin, 1985.

[225] Yong Yan, Xiaodong Zhang, and Yongsheng Song. An effective and practical performance prediction model for parallel computing on nondedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[226] John Zukowski. J2SE: New I/0. Web page, September 2004. `http://java.sun.com/developer/technicalArticles/releases/nio/`.

Index

Aperçu

Dans le domaine des systèmes distribués, la notion de mobilité du code est à l'origine de nombreux travaux visant à améliorer les performances des applications parallèles (processus légers mobiles), à faciliter le développement d'applications (agents mobiles) ou à garantir la sécurité (cartes à puces). Dans ce contexte, nous montrons que les systèmes d'agents mobiles ont peu à peu disparu au profit de plates-formes d'exécution asynchrones.

Nous présentons une nouvelle abstraction – appellée *conteneur actif* – qui est issue d'une modélisation en π-calcul d'un système d'agents mobiles, et qui semble être une brique de base avec laquelle les applications distribuées peuvent être conçues. Le développement d'une implémentation de cette abstraction en Java a fait apparaître un problème lié à la gestion de la concurrence dans les applications, distribuées ou non.

Nous décrivons donc la notion de *référence asynchrone* – notre solution à ce problème – qui permet d'exprimer simplement la concurrence d'exécution dans une application. Notre implémentation en Java de ce concept facilite le développement des applications *multi-threadées* et parallèles, en évitant le recours problématique aux *threads* par l'utilisation exclusive d'un unique paradigme : l'*appel de méthode*. Ce dernier peut se décliner en de multiples versions : synchrone, asynchrone, local ou distant.

L'ensemble de nos travaux est disponible sous licence libre LGPL au sein d'une plate-forme opérationnelle et documentée appellée Mandala qui est brièvement décrite.

Mots clés : systèmes et objets distribués, code mobile, conteneurs actifs ; concurrence, transparence, références asynchrones

---

Overview

In the domain of distributed systems, several projects focus on mobile code in order to enhance the performance of parallel applications (mobile threads), to make easier the development of applications (mobile agents) or to guarantee security (smart cards). In this context, we show how mobile agent systems have basically disappeared in favor of asynchronous execution frameworks.

We present a new abstraction – called *active container* – originating from a model of a mobile agents system. It seems to be a base layer on top of which distributed applications can be developped. A Java implementation of this abstraction raises a problem related to the management of concurrency in applications, distributed or not.

We describe the notion of *asynchronous reference* – our solution to this problem – which allows to express concurrency quite easily. Our Java implementation of this concept eases the development of multithreaded and parallel applications avoiding the problematic use of threads by the exclusive use of a single paradigm: *method invocation*. This can be: synchronous, asynchronous, local or remote.

Our work is available as an *open-source* LGPL licence package within a ready to use and documented framework called Mandala which is briefly described.

Keywords: distributed objects and systems, mobile code, active containers; concurrency, transparency, asynchronous references