

***Agents Mobiles :***  
**une implémentation en *Java* et sa modélisation.**  
**Mémoire de DEA**

par Pierre Vignéras<sup>1</sup>  
Responsable : Serge Chaumette<sup>2</sup>  
LaBRI, Laboratoire Bordelais de Recherche en Informatique,  
Université Bordeaux I,  
351 Cours de la Libération,  
33405 Talence, FRANCE

20 janvier 2000

1. Pierre.Vigneras@labri.u-bordeaux.fr  
2. Serge.Chaumette@labri.u-bordeaux.fr



## Résumé

L'objectif de ce mémoire est d'implémenter et de modéliser des agents mobiles en Java [3]. Les agents mobiles ou agents transportables, sont des codes qui se déplacent sur le réseau pour remplir une mission. Pour comprendre leur comportement et en valider des propriétés, comme par exemple leur retour sur leur machine de départ, il est nécessaire de les formaliser. La technologie Java devra être utilisée puisque ce travail s'inscrit dans le projet JEM [8]. S'il existe plusieurs implémentations différentes [26, 27, 33, 36], aucun travail de formalisation n'a été effectué à notre connaissance. Nous proposons donc une implémentation opérationnelle d'un système d'agents mobiles et sa formalisation en  $\pi$ -calcul. Nous proposons aussi une généralisation que nous appelons *conteneur actif*.

## Remerciements

Je tiens à remercier l'équipe Java du LaBRI qui m'a entouré et conseillé dans mes recherches. Notamment Asier Ugarte pour ses conseils avant chaque présentation de mes travaux en groupe de travail, Olivier Oeuillot pour son travail d'administration système et ses conseils sur la programmation en Java. Surtout, l'expérience du membre permanent de cette équipe, responsable de mon projet, Serge Chaumette pour avoir su m'orienter dans les bonnes directions lorsque plusieurs orientations possibles me laissaient perplexes.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Terminologie . . . . .	4
1.2	Applications . . . . .	5
1.2.1	Communication hétérogène . . . . .	5
1.2.2	Communications réduites . . . . .	5
1.2.3	Évolution dynamique des fonctionnalités d'un serveur . . . . .	6
1.2.4	Équilibrage de charge dynamique . . . . .	6
1.2.5	Divers . . . . .	6
1.3	But de notre travail . . . . .	6
1.4	Cadre de notre travail . . . . .	6
<b>2</b>	<b>Existant des systèmes d'agents</b>	<b>7</b>
2.1	Historique . . . . .	7
2.2	Implémentations existantes . . . . .	8
2.3	Normalisation . . . . .	8
2.4	Divers . . . . .	9
<b>3</b>	<b>Conception de notre système d'agents</b>	<b>10</b>
3.1	Implémentation . . . . .	10
3.1.1	Modèle informel . . . . .	10
3.1.2	Choix généraux . . . . .	13
3.1.3	Le serveur . . . . .	13
3.1.4	L'agent . . . . .	15
3.2	Problèmes . . . . .	17
3.2.1	Contrôle . . . . .	17
3.2.2	Destruction d'agent . . . . .	20
3.2.3	Références et copies . . . . .	20
3.2.4	Tolérance de panne . . . . .	21
3.2.5	Sécurité . . . . .	21
3.3	Évolutions possibles . . . . .	21
3.3.1	Recherche d'agents . . . . .	21
3.3.2	Broadcast de la mort d'un agent . . . . .	22
3.3.3	Chargement de code à distance . . . . .	22
3.3.4	Désactivation d'un agent . . . . .	22
3.3.5	Mort du serveur . . . . .	22
3.3.6	MobilityListener ... . . . .	23
3.3.7	Interaction dynamique et passage de message . . . . .	23
3.3.8	Tolérance de panne . . . . .	24
3.3.9	Sécurité . . . . .	24

<b>4</b>	<b>Choix d'un formalisme</b>	<b>25</b>
4.1	Le $\pi$ -calcul . . . . .	25
4.1.1	L'étude de D. Sangiorgi . . . . .	25
4.1.2	Polymorphisme . . . . .	25
4.1.3	Raffinement du typage . . . . .	25
4.1.4	Asynchronisme . . . . .	26
4.1.5	Outils . . . . .	26
4.2	CHOCS . . . . .	26
4.3	Le spi-calcul . . . . .	26
4.4	La Machine chimique abstraite . . . . .	26
4.5	Le join-calcul . . . . .	26
4.6	Les Ambiances mobiles . . . . .	27
4.7	Conclusion : choix du $\pi$ -calcul . . . . .	27
<b>5</b>	<b>Présentation du <math>\pi</math>-calcul</b>	<b>28</b>
5.1	Introduction . . . . .	28
5.2	Le $\pi$ -calcul monadique . . . . .	28
5.2.1	Syntaxe . . . . .	28
5.2.2	Congruence structurelle . . . . .	30
5.2.3	Règles de réduction . . . . .	31
5.3	Le $\pi$ -calcul polyadique . . . . .	33
5.4	Sortes . . . . .	38
5.5	Le $\pi$ -calcul d'ordre supérieur . . . . .	39
<b>6</b>	<b>Modélisation de notre système d'agents</b>	<b>41</b>
6.1	Modélisation . . . . .	41
6.1.1	Modélisation du serveur . . . . .	44
6.1.2	Modélisation de l'agent . . . . .	47
6.2	Étude de notre modèle . . . . .	47
6.2.1	L'état de l'agent . . . . .	47
6.2.2	Problème du contrôle d'un agent . . . . .	48
6.2.3	Problème de destruction d'un agent . . . . .	48
6.2.4	Problème polymorphe . . . . .	48
<b>7</b>	<b>Proposition d'une généralisation : les conteneurs actifs</b>	<b>50</b>
7.1	Le conteneur actif . . . . .	50
7.2	Implémentation . . . . .	50
7.3	Modélisation . . . . .	52
7.4	Simulation d'un système d'agent . . . . .	53
<b>8</b>	<b>Conclusion</b>	<b>56</b>

# Liste des figures

- 1.1 L'état d'un agent peut changer lors de migrations . . . . . 5
- 3.1 Représentation de l'intérieur des serveurs . . . . . 11
- 3.2 Représentation par restriction de la vue des objets du réseau . . . . . 12
- 3.3 Schéma du broadcast en  $\log_2(n)$  étapes . . . . . 22
- 3.4 Simulation de chargement du code d'un agent à distance? . . . . . 23

# Chapitre 1

## Introduction

Le monde de l'informatique a subi plusieurs bouleversements dans son histoire. L'une des plus récentes est la banalisation des réseaux et l'émergence du World Wide Web. Ces dernières révolutions en appellent d'autres comme celle des agents mobiles. Avant d'aller plus loin dans notre introduction, une terminologie est nécessaire.

### 1.1 Terminologie

#### Définition 1.1.1 (Agent)

*Un agent est un programme autonome et indépendant :*

- *autonome : l'agent n'a besoin d'aucun supplément d'information pour accomplir sa tâche;*
- *indépendant : il possède ses propres threads d'exécution.*

*Un agent peut être mobile ou immobile :*

- *mobile : il peut se déplacer (migrer) sur une autre machine pour continuer son activité;*
- *immobile : il ne peut migrer vers une autre machine.*

La migration d'un agent consiste donc à sauvegarder (sérialiser) son état - données et code, et à le restaurer (désérialiser) sur une machine distante. Cette restauration peut-être de deux types:

- *restauration bas niveau : consiste à restaurer le pointeur d'instructions de l'agent précédemment sauvegardé;*
- *restauration haut niveau : consiste à appeler une méthode particulière de l'agent.*

Le premier type de restauration pose des problèmes de sécurité. En effet, si l'état de l'agent transmis est erroné - volontairement (piraterie) ou non (mauvaise ligne) - le mauvais pointeur d'instruction restauré peut engendrer des dommages dramatiques sur l'agent qui ne reprend pas son activité normalement. Par ailleurs, l'agent n'est pas informé de son arrivée sur le serveur destination ce qui peut-être gênant. Aussi, le second type de restauration semble plus raisonnable, puisque on laisse la possibilité à l'agent de savoir si la migration a réussi en appelant une de ses méthodes.

#### Définition 1.1.2 (Serveur d'agents)

*Un serveur d'agents est le contexte d'exécution d'agents. Un agent exerce son activité sur un serveur d'agents et le destinataire d'une éventuelle migration ne peut-être qu'un serveur d'agents.*

Un agent mobile ne cesse donc de migrer de serveur en serveur et peut changer d'état lors de migrations. La figure 1.1 illustre ce phénomène en représentant l'état de l'agent par sa forme géométrique.

#### Définition 1.1.3 (Système d'agents)

*Un système d'agents est une plate-forme qui peut créer, interpréter, exécuter, transférer et terminer un agent. Il peut être un réseau de serveurs d'agents.*

Un système d'agents est la vision utilisateur de l'ensemble des serveurs et des agents sur un réseau. Aussi, dans la suite, nous confondrons souvent les termes de serveur d'agents, système d'agents et réseau de système d'agents.

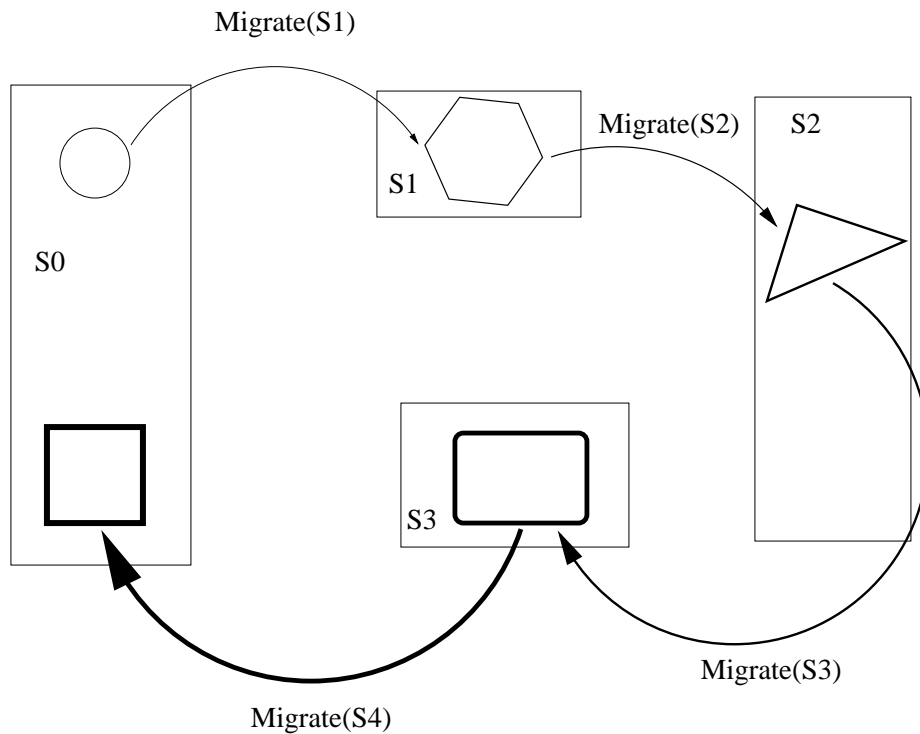


FIG. 1.1: L'état d'un agent peut changer lors de migrations

## 1.2 Applications

Les agents offrent des perspectives intéressantes. Nous allons donner une liste non exhaustive d'applications des agents et montrer ainsi leur utilités.

### 1.2.1 Communication hétérogène

Bien que les réseaux apportent beaucoup d'avantages, la multiplicité des systèmes augmente l'incompatibilité des langages de commandes et des formats de données. Les agents peuvent résoudre ce problème sous contrainte que les primitives du langage de l'agent soit exécutable sur tout système.

En effet, un client nécessite un ensemble de services proposé par un serveur. Ces services consiste généralement en un ensemble d'appels systèmes différents d'un système à l'autre. La solution consiste à utiliser un agent qui traduira le service en une suite d'appels systèmes une fois la migration sur le serveur prestataire effectuée.

### 1.2.2 Communications réduites

Les agents peuvent réduire les ressources utilisées en communications.

Sur un serveur, lorsqu'une requête nécessite le calcul d'une image à partir de données brutes (statistiques d'accès par exemple), il peut-être plus efficace d'envoyer un agent au client qui transporte les données et effectue le calcul de l'image une fois migré. Mieux, l'agent en fournissant une réelle application, permet à l'utilisateur de modifier les données, de les représenter sous diverses formes sans intervention du serveur.

Inversement, envoyer l'agent du client au serveur réduit le coût des communications d'un client connecté par une mauvaise ligne à un fournisseur d'accès Internet (modem par exemple). En effet, lors du téléchargement d'un gros fichier, le client après avoir envoyer son agent peut se déconnecter. L'agent côté serveur utilise le temps nécessaire pour télécharger de gros fichiers et les stocker dans un répertoire temporaire connu à l'avance. Enfin, le client peut récupérer ses fichiers au maximum du débit de son modem.

### 1.2.3 Évolution dynamique des fonctionnalités d'un serveur

Un agent après avoir migré est considéré comme une nouvelle fonctionnalité du serveur destination. Les clients communiquent avec l'agent directement selon un protocole qui peut ne pas être connu du serveur.

### 1.2.4 Équilibrage de charge dynamique

En parallélisme, dans les machines de type MIMD, l'enjeu principal dans l'écriture d'un programme efficace est l'équilibrage de charge. Cette équilibrage est difficile à résoudre sur des problèmes très irréguliers (traçage de rayons par exemple). Les agents mobiles permettent d'équilibrer la charge d'une machine : si un noeud est en avance sur son travail tandis qu'un autre est en retard, un ou plusieurs agents peuvent migrer du dernier au premier réduisant ainsi l'écart.

### 1.2.5 Divers

Il existe beaucoup d'autres applications des agents mobiles. Nous pouvons par exemple imaginer la notion d'agent vendeur et d'agents acheteurs. Un agent vendeur et un agent chargé de vendre un produit qui voyage de serveur en serveur à la recherche d'agents acheteurs de ce même produit. Une fois une rencontre effectuée, un algorithme de marchandage peut entrer en fonction. Si un accord d'achat est trouvé, il est réalisé directement (l'agent acheteur possède une cyber-monnaie) ou/et un courrier électronique est envoyé aux propriétaires de ces agents.

Enfin, des compétitions peuvent être imaginées entre agents. Considérons par exemple un ensemble de serveur dont un est nommé serveur de départ, et un autre serveur d'arrivée. Chaque agent doit migrer de serveur en serveur en partant du départ et en terminant à l'arrivée. Sur chaque serveur, les agents doivent effectuer une tâche spécifique (par exemple, calculer une image en trois dimensions). Le premier arrivé à gagné.

## 1.3 But de notre travail

Nous avons vu quelques applications des systèmes d'agents. L'abondante littérature les concernant montre qu'une révolution technologique est en marche. Cette littérature concerne soit les implémentations existantes - nous en donnerons quelques-unes dans le chapitre 2, soit les modèles de calculs de processus mobiles que nous verrons en 4. Toutefois, il n'existent pas, à notre connaissance, de formalisation des agents mobiles. Aussi, après l'étude de l'implémentation de notre système d'agents (chapitre 3.1.2), nous la formaliserons au chapitre 6 en  $\pi$ -calcul - un modèle de calcul que nous introduirons au chapitre 5. Enfin, notre étude débouchera (chapitre 7) sur l'étude d'un nouvel objet appelé *conteneur actif* plus général qu'un système d'agents qui permet - entre autres - leur simulation.

## 1.4 Cadre de notre travail

Notre étude s'inscrit dans un projet beaucoup plus général baptisé JEM [8] (Expérimentation environMent for Java) développé par l'équipe Java du LaBRI et dirigé par Serge Chaumette. Il consiste à :

- faciliter l'usage et la programmation de systèmes distribués hétérogènes grâce à l'implémentation d'une plateforme distribuée. Java permet l'hétérogénéité de notre plateforme tandis que la distribution est gérée par les technologies RMI [20] ou CORBA [40].
- utiliser la plate forme ci-dessus pour étudier, avec d'autres équipes de recherche, des problèmes difficiles liés au parallélisme et aux technologies distribuées : le débogage d'applications parallèles, les threads distribuées et les agents mobiles.

Ce cadre justifie l'utilisation du langage Java pour implémenter un système d'agents mobiles. En effet, notre implémentation doit s'intégrer au projet JEM.

Pourtant, il existe un langage dédié aux agents mobiles [16]. Basé sur le langage dérivé de CCS [21] : Facile [44], il fournit des primitives pour la migration de code. Toutefois, notre cadre de travail étant fixé, il ne s'agit pas de modéliser les agents mobiles, mais de modéliser une implémentation en Java d'un système d'agents mobiles.

## Chapitre 2

# Existant des systèmes d'agents

Ce chapitre tente de donner une idée de l'état actuel des connaissances en matière d'agents. Cette tâche est assez difficile pour plusieurs raisons:

- Cette technologie est en pleine essor bien que la notion d'agent soit assez ancienne comme nous le verrons en 2.1.
- Il existe beaucoup d'implémentations différentes de la notion d'agent. La section 2.2 en décrira quelques-unes.
- Il y a eu plusieurs tentatives de normalisation pour prévenir la profusion d'implémentations incompatibles entre elles. Ces normalisations sont l'objet de la section 2.3.
- Il y a plusieurs modèles pour décrire le comportement des agents. Chacun fait l'objet de recherches et de documents qui ne permettent pas d'avoir une vue globale et homogène. Nous en verrons quelques-uns en 4.
- Le site de l'Agent Society [35] - qui a vu le jour en 1996 - s'est chargé de réunir toutes les informations sur les agents. Cependant, malgré les efforts effectués, l'évolution très rapide du domaine ne permet pas d'avoir une information toujours pertinentes et actuelles.
- La maîtrise technologique des agents est un enjeu économique et stratégique très important, ce qui explique la difficulté de trouver une normalisation acceptable. En effet les entreprises qui ont déjà implémenté leur système d'agents ont du mal à se mettre d'accord sur un standard.
- La technologie des agents pose des problèmes théoriques et pratiques, notamment en terme de sécurité, très difficiles et dont l'étude en est encore au stade de la recherche.

Toutes ces raisons expliquent pourquoi obtenir une idée précise de l'état actuel des technologies est une gageure. Par exemple, au début de nos recherches, il existait principalement trois sociétés connues pour leurs système d'agents : IBM, Mitsubishi et General Magic. Entre-temps beaucoup de produits ont vu le jour.

## 2.1 Historique

La notion de code mobile a été introduite et implémentée pour la première fois dans un réseau hétérogène d'ordinateurs en 1972-1974 [37]. Cette idée a donné naissance à un ordinateur distribué de 256 processeurs. Produit industriellement en série et amélioré, il a été utilisé - entre autres - pour la modélisation en aérodynamique.

Toutefois, le nom d'agent mobile a été inventé par General Magic [28] dans leur système d'agents Telescript. Ce système était très avancé et offrait plus de fonctionnalités que son remplaçant Odyssey [33] écrit entièrement en Java. Ce remplacement est justifié par l'adoption générale du langage Java pour les applications de haut niveau multi-plateforme et par l'apparition récente de nombreux concurrents.

Plusieurs raisons peuvent expliquer l'engouement soudain pour cette technologie.

D'une part, les technologies réseau - en plein essor depuis quelques années seulement - rendent la généralisation des agents mobiles envisageable. La vitesse des transmissions, la sécurité des communications et la standardisation des protocoles rendent plus facile leur mise en œuvre. D'autre part, l'accroissement du pouvoir d'expression des calculs

formels (cf section 4) autorise la modélisation d'architectures distribuées indispensable à la notion d'agents mobiles. Enfin, les langages de haut niveau (Java, C++, Facile, Pict, Obliq) permettent l'écriture rapide de système d'agents.

## 2.2 Implémentations existantes

Il existe à l'heure actuelle beaucoup d'implémentations différentes incompatibles entre elles. Certaines écrites en Java *pur* (ce qui est aussi noté 100% Java), bénéficient de fait du portage multi-plateformes. Citons donc d'abord quelques systèmes écrits en Java :

- Odyssey ([33]) de General Magic, successeur de Telescript, il en reprend les concepts fondamentaux à savoir les notion de places, tickets et autorités. De plus, il est indépendant du protocole de transport utilisé et peut donc s'adapter à RMI (Sun), CORBA (OMG) ou DCOM (Microsoft). Odyssey est téléchargeable gratuitement pour le développement d'applications non commerciales.
- Les Aglets ([26]) d'IBM, utilisent la notion de référence distante (AgletProxy), le passage de messages (multicast, broadcast) et le protocole ATP développé par IBM (Agent Transfert Protocol [18]). Il offre en outre une certaine notion de sécurité en limitant les ressources allouées aux aglets. Le lecteur intéressé se référera à [23] pour les spécifications et à [19] pour le manuel utilisateur.
- Concordia ([27]) de Mitsubishi, est un système d'agents similaire aux deux précédents. Le code est téléchargeable pour le développement d'applications non commerciales.
- Voyager ([36]) de ObjectSpace semble être le système d'agent le plus complet puisqu'il intègre entre autres les références distantes, le support CORBA, et un ramasse miettes distribué.
- Sumatra ([1]) est une extension du langage Java. Sa principale caractéristique est d'autoriser dynamiquement l'allocation de ressources à des objets mobiles. Il n'est pas à proprement parler un système d'agents.

Enfin, les systèmes suivants non écrit en Java peuvent être digne d'intérêt :

- Agent Tcl (Dartmouth)
- April (Fujitsu)
- Clearlake (Guideware)
- M0 (Univ. of Geneva)
- Obliq (DEC SRC)
- Tacoma (Univ. of Norway & Cornell Univ.)
- Telescript (General Magic)
- Wave (Univ. of Surrey)

Malheureusement, un agent écrit dans un de ses systèmes ne peut migrer vers un système différent. C'est pourquoi plusieurs tentatives de standardisation ont vu le jour.

## 2.3 Normalisation

La première tentative de normalisation est originaire d'une réunion entre IBM et General Magic ([46]). Après plusieurs désaccord et plusieurs ébauches de normalisation ([10], [11], [14]) un colloque ([2]) a été organisé en février 1997 pour établir une norme. Finalement, le 10 janvier 1997, l'OMG [32] (Object Management Group) publie une spécification - la MASIF (Mobile Agent System Interoperability Facilities Specification) - qui interrompt définitivement les querelles entre les développeurs de systèmes d'agents. En effet, l'OMG est populaire pour son système d'architecture distribué appelé CORBA qui s'affranchit des barrières des systèmes d'exploitation, des protocoles et des langages utilisés. C'est donc en utilisant ce système qu'il est possible de rompre avec l'incompatibilité des systèmes d'agents.

Malheureusement, cette spécification est très lourde, et il n'existe, à notre connaissance, aucun système d'agents qui la supporte intégralement.

## 2.4 Divers

Nous pouvons remarquer les conférences internationales annuelles sur les applications pratiques des agents et multi-agents intelligents : PAAM (Practical Applications of Intelligent Agents and Multi-Agents). Le lecteur intéressé pourra se référer au site qui héberge leurs comptes-rendus([29]).

Depuis quelques années, une abondante littérature émane du Web sur la notion générale de code mobile. Elle peut concerner soit une liste d'implémentations existantes, par exemple la page HTML [31] ou une liste de bibliographie sur les calculs de processus concurrents qui modélise la notion de code mobile ([30] et [34]).

Les prochains chapitres se consacrent à l'objet de ce mémoire : la modélisation des agents mobiles en Java.

# Chapitre 3

## Conception de notre système d'agents

Nous l'avons vu, les agents offrent des perspectives très intéressantes et une nouvelle vision des réseaux. Il existe plusieurs implémentations incompatibles entre elles et une tentative de normalisation a du voir le jour. Toutefois, à notre connaissance, aucun modèle théorique n'a été présenté jusqu'à présent.

Pour arriver à trouver un modèle, un compromis doit être réalisé. Faut-il adapter le modèle à un formalisme précis ou bien choisir un formalisme en fonction du modèle ? Il n'est pas évident de répondre à cette question. En effet, en plus du modèle théorique, une implémentation doit être faite dans un langage défini à l'avance puisque ce projet s'inscrit dans un autre plus général dont la colonne vertébrale est liée à un langage particulier : Java. Nous avons vu ce point en 1.4. Par conséquent, un modèle informel a été élaboré, puis implémenté dans le langage cité. C'est dans une dernière étape que le modèle formel a été écrit. Enfin, des ajustements ont été fait régulièrement entre l'implémentation et le modèle formel.

Par conséquent, notre modèle informel et son implémentation en Java sont présentés dans ce chapitre. Nous étudions aussi les problèmes que posent notre implémentation (section 3.2) et ses possibilités d'évolutions (section 3.3).

### 3.1 Implémentation

#### 3.1.1 Modèle informel

Une première approche consiste à se demander ce qu'est un système d'agents et comment le représenter. Une représentation "classique" est schématisée sur la figure 3.1. Sur celle-ci, un serveur d'agents *contient* des agents. Par exemple, le serveur  $S_1$  contenait les agents  $A_1, A_2$  et  $A_3$  juste avant que ce dernier migrer sur le serveur  $S_2$ .

Considérons maintenant le réseau tout entier. Sans se préoccuper de la notion d'intérieur d'un serveur d'agents. Nous avons donc un ensemble d'objets de type différent (Agent et Serveur) qui s'exécutent en *concurrency*. Un réseau de serveur d'agents est comme tout réseau représentable sous forme de graphe. Toutefois, si le graphe 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'importances, les données ne sont pas représentées, en général. Dans notre cas, il n'en est pas de même, puisque notre représentation doit rendre compte de l'activité des agents. Il faut donc représenter de manière simple - pour l'implémenter et la modéliser - la notion d'intérieur d'un serveur.

Lorsqu'un agent est dans un serveur, nous dirons qu'il est local à ce serveur. Le serveur à un accès privilégié sur lui : 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 schématisé sur la figure 3.2. 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 objets du réseau. La vue d'un objet est l'ensemble des objets qui lui sont accessibles<sup>1</sup>. Évidemment, un objet appartient à sa propre vue.

---

1. Un objet a accès à un autre s'il possède une référence directe sur lui.

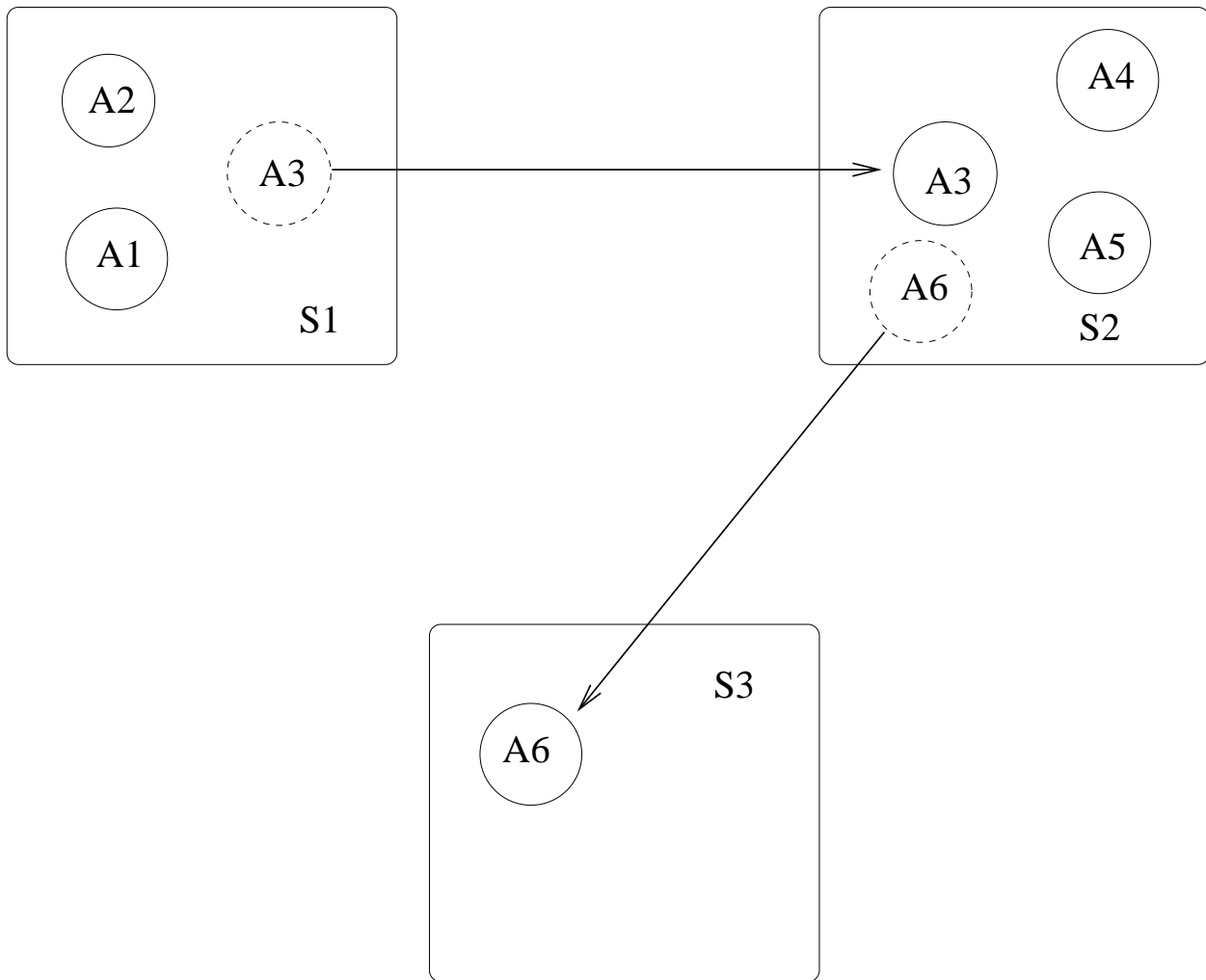


FIG. 3.1: Représentation de l'intérieur des serveurs

Le *serveur local* d'un agent est le seul objet 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.

La dernière de ces conséquences peut sembler trop restrictives. En effet, si les agents ne peuvent communiquer entre eux, l'intérêt d'un tel modèle paraît limité. Heureusement, il existe une solution à ce problème. Rien n'interdit en effet les références indirectes. Nous appellerons *identificateur local* (ABRÉV. *id*) une telle référence indirecte. Un *id* devra être unique pour chaque agent d'un serveur et être fourni à tout objet souhaitant communiquer avec un agent. Nous pourrions 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.

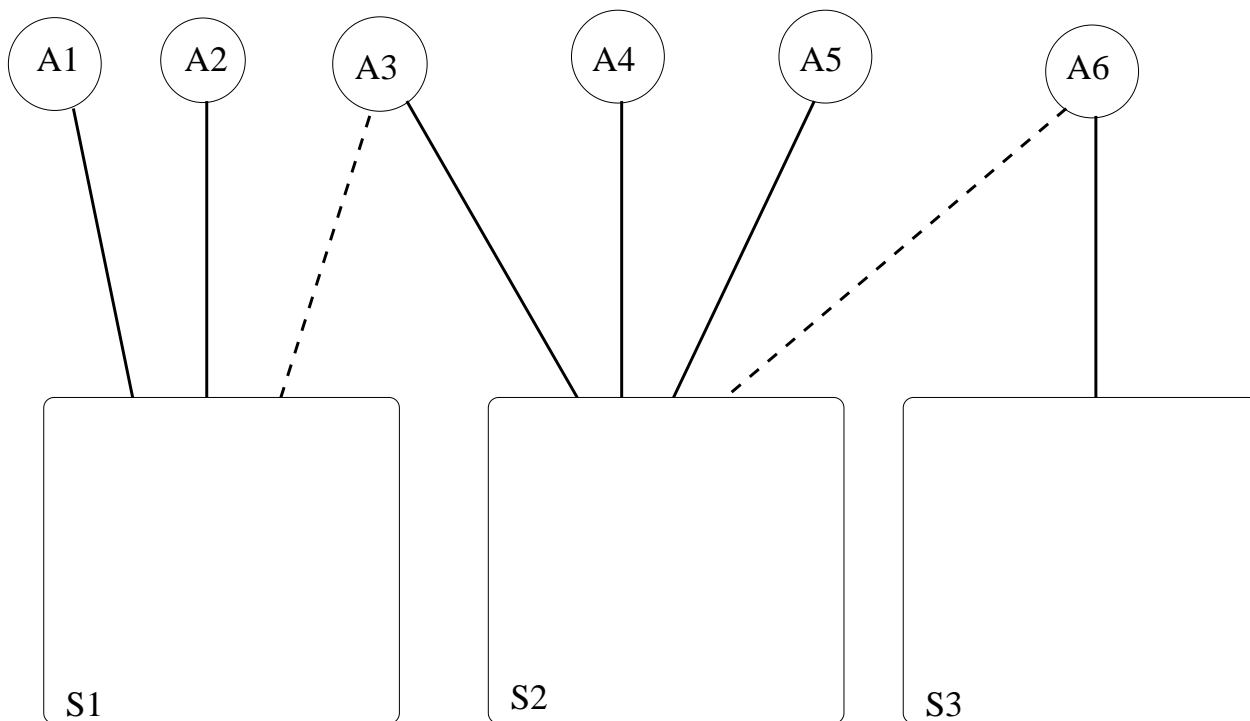


FIG. 3.2: *Représentation par restriction de la vue des objets du réseau*

Les sections suivantes de ce chapitre présentent notre implémentation d'un système d'agents. Dans un premier temps, nous détaillons les choix qui ont dû être faits et leur implication sur le logiciel. Ensuite, une étude approfondie du code sera suivie d'une discussion sur les problèmes que posent notre implémentation.

### 3.1.2 Choix généraux

Comme nous l'avons vu en 3.1.1, un agent n'existe que s'il est lié à un serveur local. La notion de serveur implique la notion de client et finalement la notion de protocole. Le protocole utilisé aurait pu être :

- un protocole dédié à notre application, une telle entreprise aurait eu beaucoup d'influence sur le temps de débogage et aurait alourdi le code;
- le protocole ATP d'IBM présenté en [18], mais il est essentiellement utilisé par IBM;
- CORBA qui est très utilisé en ce moment mais relativement lourd à mettre en oeuvre.

Finalement, nous avons préféré utiliser un protocole dédié à la programmation Java distribué : RMI. Ce protocole est relativement simple à utiliser, entièrement destiné à Java et utilisé dans la plateforme JEM qui est, rappelons-le, le cadre de notre travail. Toutefois, le passage à CORBA ne devrait pas poser de problème particulier.

Par ailleurs, le serveur ne fournit aucun système de sécurité. Son objectif est d'être simple pour faciliter la modélisation. Néanmoins, sa conception le rend relativement évolutif ce que nous verrons en 3.3.

Malgré tout, cette implémentation possède deux spécificités qui ne sont pas mentionnées dans notre modèle informel (ni dans sa formalisation d'ailleurs). La première concerne la gestion des *id* particulières puisque lorsqu'un agent visite plusieurs fois un même serveur, il conserve son *id* sur ce serveur. Ainsi, la référence - i.e l'*id* - d'un agent qui a migré devient invalide jusqu'à ce que l'agent revienne sur le serveur. Nous appellerons cette propriété la *pseudo-validité référentielle* et verrons comment cette fonctionnalité est implémentée en 3.1.3 côté serveur et en 3.1.4 côté agent. Enfin, le paragraphe 3.3.1 étudiera les propriétés intéressantes qu'elle possède. La deuxième que nous verrons en 3.1.4 concerne la *tolérance de panne*. Lorsqu'un serveur tombe en panne et doit être relancé, le réseau reste cohérent. Notamment, les références au serveur remis en service sont valides. Par contre, nous verrons les problèmes que cette tolérance entraîne en 3.2.4.

### 3.1.3 Le serveur

Nous allons donner un aperçu du code du serveur. Pour assurer la pseudo-validité référentielle, le serveur doit garder dans une table la référence des agents dans son serveur. Ainsi, lors de la migration d'un agent, le serveur remplace dans sa table l'agent par une copie. Cette copie sera nommée *image invalidée*.

Voyons tout d'abord l'interface donnée dans le fragment de code 3.1.1. Le protocole RMI est utilisé et c'est la raison pour laquelle `AgentServer` étend `java.rmi.Remote`. Pour la même raison, toutes les méthodes de `AgentServer` doivent déclarer le lancement possible de l'exception `java.rmi.RemoteException`. C'est l'interface `AgentServer` qui est visible aux objets extérieurs. En réalité, c'est la référence à la classe qui implémente cette interface à savoir `AgentServerImpl`. Nous parlerons de cette dernière en 3.2.

Les deux méthodes `sendAgent` et `receiveAgent` ne doivent pas être utilisées par un autre objet qu'un `AgentServer`. Elles réalisent respectivement l'envoi et la réception d'un agent. L'envoi d'un agent ne pose pas de problème particulier : après avoir appelé la méthode `onMigrating`, la méthode `receiveAgent` du serveur destinataire est appelée et tous les threads de l'agent sont arrêtés. Enfin, l'agent est remplacé dans la table du serveur par son image invalidée. La réception doit allouer un nouvel *id* pour l'agent reçu sauf s'il a déjà visité le serveur. Dans ce cas, l'*id* précédemment alloué lui est redonné, garantissant ainsi la pseudo-validité référentielle. L'agent est enregistré dans la table (écrasant son image invalidée si nécessaire). Enfin, sa méthode `onMigration` est appelée avant de lui allouer un nouveau thread pour l'exécution de sa méthode `run`.

Une autre méthode est réservée au serveur : `agentKilledBy`. Lorsqu'un agent est détruit sur le réseau, son identificateur local doit être libéré. C'est l'objet de cette méthode. Si la méthode `killAgent` d'un serveur est appelée, ce dernier, après avoir effectivement détruit l'agent dont la référence est passée en paramètre, appelle la méthode `agentKilledBy` de tous les serveurs que l'agent a visités (broadcast). Une implémentation plus efficace de cette méthode sera présentée en 3.3.2.

La méthode `createAgent` réalise la création d'un agent et son activation - i.e l'allocation d'une thread pour sa méthode `run`. Une extension de cette méthode sera vue en 3.3.3.

---

**Fragment de code java 3.1.1 L'interface AgentServer**

---

```
package agents.model;

public interface AgentServer extends java.rmi.Remote {
    /** This method is used by AgentServer objects exclusively.
     * Return the localID that agent receive. */
    public String receiveAgent(Agent agent) throws java.rmi.RemoteException;

    /** This method is used by AgentServer objects exclusively.
     * Send an Agent object to an AgentServer object located at address
     * and return the localID the Agent receive on it. */
    public String sendAgent(String localID, String targetServerAddress)
        throws java.util.NoSuchElementException, InvalidAgentException,
            TargetServerRemoteException, java.rmi.RemoteException;

    /** This method is used by AgentServer objects exclusively. */
    public void agentKilledBy(String serverAdress, String localID)
        throws java.rmi.RemoteException;

    /** Returns a LocalID allowing remote process to communicate
     * with the associated Agent with a call to callAgentMethod(). */
    public String createAgent(java.net.URL codeBase, Object arg)
        throws ClassNotFoundException, IllegalAccessException,
            IllegalAccessException, java.rmi.RemoteException;

    /** Try to kill the agent with identifiicator localID.
     * It's death is thrown to each server the agent has visited. */
    public void killAgent(String localID)
        throws java.util.NoSuchElementException, java.rmi.RemoteException,
            InvalidAgentException;

    /** Allows remote objects to call an Agent method. */
    public Object callAgentMethod(String localID, String methodName, Object args[])
        throws java.util.NoSuchElementException, InvalidAgentException,
            IllegalAccessException, IllegalArgumentException,
            java.lang.reflect.InvocationTargetException, NoSuchMethodException,
            SecurityException, java.rmi.RemoteException;

    /** Return a copy of the Agent with identifiicator localID.
     * WARNING: the copy is not an agent since it is not hosted
     * by an AgentServer. It cannot migrate and will throw a
     * NullPointerException. */
    public Agent getAgentCopy(String localID)
        throws java.util.NoSuchElementException, java.rmi.RemoteException;

    public java.util.Hashtable getAllAgentsCopy() throws java.rmi.RemoteException;

    public java.util.Enumeration getAllLocalID() throws java.rmi.RemoteException;

    /** Return the host adress of this AgentServer object. */
    public String getHost() throws java.rmi.RemoteException;

    /** Return the host adress of this AgentServer object
     * and the protocol use in URL syntax. */
    public String getServerAddress() throws java.rmi.RemoteException;

    /** Make this AgentServer down.
     * All dispose() method of any agent active on this server
     * will be called. */
    public void exit() throws java.rmi.RemoteException;
}
```

Le lecteur attentif aura remarqué que les méthodes **réservées** au serveur sont déclarées **publiques**. Cela est **nécessaire** pour permettre à un serveur distant d'appeler une de ces méthodes. Nous touchons là un problème important : celui du contrôle de l'agent. Nous l'analyserons en 3.2.1.

Quelques méthodes sont des services non indispensables mais qui peuvent faciliter l'écriture d'une interface pour le contrôle du serveur, voire l'écriture d'agents (`getAgentCopy` par exemple). Elles n'auront naturellement pas d'équivalent dans le modèle formel et posent elles aussi des problèmes conceptuels que nous verront en 3.2.3.

Enfin, la méthode `exit` permet au serveur de cesser son activité. Avant de terminer, il appelle la méthode `dispose` de tous les agents valides dont il est le serveur local. Une évolution de cette méthode sera vu en 3.3.5.

### 3.1.4 L'agent

La classe agent est relativement simple. Elle est déclarée dans le paquetage `agents.model` comme le montre le fragment de code 3.1.2.

---

#### Fragment de code java 3.1.2 Déclaration de la classe Agent

---

```
package agents.model;

import java.io.*;
import java.net.*;
import java.rmi.*;

import util.*;

/** A generic agent. This class should be overridden to
    be able to do something.
    WARNING : constructors can only be called by an
    AgentServer object ! */

public class Agent implements Serializable, Modifiable {
    public static final boolean isModifiableInitStatus = true;
    private String originalServerAddress;
    private String fromServerAddress;
    private String currentServerAddress;
    private AgentID id;
    private boolean isMigrable = true;
    private boolean isModifiable = isModifiableInitStatus;
    private boolean isValid = false;
    private transient ThreadGroup agentThreadGroup = null;
    ...
}
```

---

Cette classe implémente l'interface `java.io.Serializable` pour autoriser la migration de son code. L'interface `Modifiable` que nous avons définie autorise la modification du statut *migrable* de l'agent. Si un agent est marqué comme non migrable alors toute tentative de migration génère une exception.

Parmi ses champs, deux sont intéressants. Le premier déclare un objet *id* de type `AgentID`. C'est la référence de l'agent. Plus exactement, c'est une table de correspondance entre un serveur visité et l'*id* de l'agent sur ce serveur. Les méthodes `getLocalID` et `getAllServers` de la classe `AgentID` renvoient respectivement l'*id* de l'agent sur le serveur passé en paramètre et l'ensemble des serveurs visités par l'agent. C'est en utilisant cette classe que la propriété de pseudo-validité référentielle est maintenue.

Pour assurer la tolérance de panne, il est nécessaire de ne pas faire correspondre via le champ *id* un *id* local à un objet - au sens programmation orienté objet - de type `AgentServer`. En effet, la référence à un objet RMI n'est valide que durant la vie de l'objet. S'il est détruit (panne du serveur), et qu'un nouveau est créé (relancement du serveur), la référence est invalide. Aussi, nous utilisons comme référence à un serveur, son adresse au sens RMI. C'est une chaîne de caractère de type URL : `rmi://machine:port/nom` où *nom* est le nom attaché à l'objet.

Quelques méthodes sont réservées au serveur, notamment la méthode `__initAgent` qui permet l'initialisation correcte des champs de l'agent.

Deux "inner-class" permettent aux méthodes `dispose` et `migrate` de fonctionner correctement. En effet, ces deux méthodes doivent arrêter les threads allouées à l'agent. Il faut donc que cet arrêt soit effectué dans une thread parente. C'est pourquoi ces deux méthodes allouent une thread et l'attache au ThreadGroup père. Ces deux threads sont chargées d'appeler les méthodes adéquates du serveur local comme le montre le fragment de code 3.1.3. Un artifice est utilisé pour contourner la différence de déclaration entre la méthode `run` de l'interface `java.lang.Runnable` et la méthode `sendAgent` de la classe `AgentServer` qui renvoi un *id* et est susceptible de renvoyer une exception. Cela n'est pas le cas avec la méthode `dispose` puisque la référence au serveur local est toujours valide.

---

**Fragment de code java 3.1.3** Les deux "inner-classes" de la class `Agent`

---

```
private class MigrateThread implements Runnable{
    private String targetServerAddress;
    private Exception exception = null;
    private String newLocalID = null;

    public MigrateThread(String targetServerAddress){
        this.targetServerAddress = targetServerAddress;
    }
    public void run(){
        try{
            String myLocalID = id.getLocalID(currentServerAddress);
            newLocalID =
                ((AgentServer) Naming.lookup(currentServerAddress)).sendAgent(myLocalID,
                                                                              targetServerAddress);
        } catch(Exception e){
            this.exception = e;
        }
        synchronized(this$0){
            this$0.notify();
        }
    }
    public Exception getException(){
        return exception;
    }
    public String getNewLocalID(){
        return newLocalID;
    }
}
private class DisposeThread implements Runnable{
    public void run(){
        try{
            ((AgentServer) Naming.lookup(currentServerAddress)).killAgent(id.getLocalID(currentServerAddress));
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

---

Le code des deux méthodes `migrate` et `dispose`, déclarées `final` pour éviter la surcharge est donné en 3.1.4.

Les méthodes `onCreation`, `onMigration`, `onMigrating`, `onDisposing` et `run` sont vides et héritables ce qui permet la création d'agents "utiles". Enfin, d'autres services plus généraux comme les méthodes `getCurrentServerAddress`, `getFromServerAddress`, `getOriginalAddress` facilitent l'écriture et la mise au point des agents.

---

**Fragment de code java 3.1.4** Méthodes migrate et dispose de la classe Agent

---

```
/** WARNING: dispose() cannot kill all threads allocated by this
agent. Use isValid() to terminate properly */ public synchronized
final void dispose(){ new Thread(agentThreadGroup.getParent(), new
DisposeThread()).start(); }

/** Try to send this Agent to the AgentServer located at targetServerAddress.
If this agent is UnMigrable, an UnMigrableObjectException is thrown.
If an error occurs during migrating, an Exception is thrown.
If migrate() is succesfull, the isValid() will return false allowing
any threads allocated to this Agent to terminate properly.
Return the locaID of this agent on the targetServerAddress.
*/
public synchronized final String migrate(String targetServerAddress)
throws UnMigrableObjectException, Exception {
    if (!isMigrable){
        throw new UnMigrableObjectException();
    }
    MigrateThread migrateThread = new MigrateThread(targetServerAddress);
    Thread thread = new Thread(agentThreadGroup.getParent(), migrateThread);
    thread.start();
    wait();
    Exception e = migrateThread.getException();
    if (e != null){
        throw e;
    }
    return migrateThread.getNewLocalID();
}
```

---

## 3.2 Problèmes

Plusieurs problèmes ont été soulevés, il est temps soit de les résoudre, soit d'en comprendre l'origine.

### 3.2.1 Contrôle

Dans notre implémentation, rien n'empêche un agent de lancer son propre thread pour l'exécution de la méthode `run` dans l'une de ses méthodes *incontrôlables* i.e les méthodes `onCreation`, `onMigration`, `onMigrating`, `onDisposing` et `run`. Le serveur allouant automatiquement un thread pour la méthode `run` se retrouve avec deux threads alloués pour cette méthode au lieu d'un. Considérons en effet un agent dont la méthode `onMigration` est défini selon le fragment de code 3.2.1.

---

**Fragment de code java 3.2.1** Méthodes incontrôlables

---

```
public void onMigration(){ ... new Thread(this).start() }
```

---

A chaque migration, l'agent qui arrive sur un serveur alloue un thread pour sa méthode `run` via l'appel automatique de sa méthode `onMigration` par le serveur. Et un nouveau thread est alloué pour la même méthode `run` par le serveur pour - normalement - activer l'agent. Il est clair que cela n'est pas raisonnable puisque un agent peut créer autant de thread qu'il le souhaite dans ses méthodes *incontrôlables*. Le terme de *incontrôlables* pour les méthodes précédentes est justifié par l'incapacité d'un serveur de savoir ce que chacune de ces méthodes peut effectuer.

Par ailleurs, il est nécessaire d'assurer la séquentialité de l'exécution des méthodes `onCreation` et `run`, `onMigration` et `run` sans entraver le serveur. Par exemple, un agent peut exécuter une boucle infinie dans `onCreation`. Si cette méthode est appelée dans le même thread que l'appel à `createAgent`, alors ce dernier ne termine jamais comme le montre le fragment de code 3.2.2. Il en va de même avec toutes les méthodes de l'agent *incontrôlable* (`onCreation`, `onMigration`, `onMigrating`, `onDisposing`, `run`). Il est donc nécessaire d'allouer une thread pour chaque

appel à une méthode incontrôlable comme le montre le fragment de code 3.2.3.

---

**Fragment de code java 3.2.2 Appel d'une méthode incontrôlables sans allocation de thread**

---

```
public synchronized String createAgent(URL codeBase, Object arg)
    throws ClassNotFoundException,
           IllegalAccessException,
           RemoteException {
    Class a;
    String localID = null;
    try{
        // is codeBase local ?
        if (...) {
            // Yes => create the agent
            String slashFile = codeBase.getFile();
            String javaClassName = slashFile.substring(slashFile.lastIndexOf("/") + 1);
            a = Class.forName(javaClassName);
            // Check if this class extends Agent class
            if (...){
                //No
                throw new IllegalAccessException();
            }
            //Yes
            AgentID id = createNewAgentID();
            localID = id.getLocalID(address);
            Agent agent = (Agent) a.newInstance();
            agent.__initAgent(address, id);
            agentsHashtable.put(localID, agent);
            //Problem here if onCreate doesn't return.
            agent.onCreate(arg);
            //Allocate a new thread for agent.run() method
            startAgentThread(new AgentSafeThread(agent));
            return localID;
        }
    } catch(Exception e){
        e.printStackTrace(System.err);
        return null;
    }
    // No => throw an Exception.
    throw new ClassNotFoundException();
}
```

---

Malgré tout, allouer une thread pour les méthodes `onMigrating` et `onDisposing` semble être contradictoire avec l'objectif final de ces deux méthodes : désactiver l'agent i.e terminer ses threads. Ce paradoxe est difficile à résoudre. Nous verrons une solution en 3.3.4.

En outre, un agent peut appeler ses propres méthodes incontrôlables. Plus grave encore, tout objet connaissant l'*id* local d'un agent peut appeler ses méthodes publiques dont certaines devraient être réservées.

Nous pourrions penser pour résoudre ce problème à un langage offrant une protection des méthodes plus fines. Plutôt que d'être globalement publiques, les méthodes seraient publiques à un certain nombre de classes et pas à d'autres. Or, non seulement le langage Java possède cette richesse d'expression avec la notion de package, mais surtout cela ne résoudrait le problème que dans le cas statique. En effet, Un agent peut autoriser dynamiquement certains objet à interagir avec lui. Une solution sera proposée en 3.3.7. Notre implémentation se voulant simple, ne propose pas de solution.

Un dernier point sur le contrôle vient des méthodes publiques du serveur qui peuvent être appelées par n'importe quel objet. Cela est nécessaire pour permettre par exemple à une interface de communiquer avec le serveur. Cette interface, destinée à l'administrateur du système d'agents, doit pouvoir être écrite par un tiers, et utiliser toutes les fonctionnalités du système d'agent. Ce problème peut cependant être résolu comme nous le verrons en 3.3.9.

---

**Fragment de code java 3.2.3 Allocation de thread pour chaque appel d'une méthode incontrôlables**

---

```
public synchronized String createAgent(URL codeBase, Object arg)
    throws ClassNotFoundException,
        IllegalAccessException,
        RemoteException {
    Class a;
    String localID = null;
    try{
        // is codeBase local ?
        if (...) {
            // Yes => create the agent
            String slashFile = codeBase.getFile();
            String javaClassName = slashFile.substring(slashFile.lastIndexOf("/") + 1);
            a = Class.forName(javaClassName);
            // Check if this class extends Agent class
            if (...){
                //No
                throw new IllegalAccessException();
            }
            //Yes
            AgentID id = createNewAgentID();
            localID = id.getLocalID(address);
            Agent agent = (Agent) a.newInstance();
            agent.__initAgent(address, id);
            agentsHashtable.put(localID, agent);
            // Call onCreation Agent's method and run in a new allocated thread.
            startAgentThread(new AgentSafeThread(agent, arg));
            return localID;
        }
    } catch(Exception e){
        e.printStackTrace(System.err);
        return null;
    }
    // No => throw an Exception.
    throw new ClassNotFoundException();
}
```

---

### 3.2.2 Destruction d'agent

Le langage Java ne fournit pas de fonctionnalités pour éliminer un objet de la machine virtuelle. La seule solution est d'éliminer toutes les références à cet objet. Cette solution n'est pas envisageable dans la mesure où un agent peut créer plusieurs objets inconnus du serveur qui le référence. Il peut donc y avoir une accumulation d'objets en mémoire. De plus, il n'est pas possible en Java d'arrêter une thread sans son accord. La méthode `stop` de la classe `java.lang.Thread` n'est pas sûre. Le lecteur désireux d'en savoir plus pourra se reporter à l'API du jdk 1.2 de Sun. Disons seulement que c'est la raison pour laquelle cette méthode (entre autres) a été dépréciée dans la nouvelle version du jdk. Il est donc déconseillé de l'utiliser. Aussi, faut-il trouver une solution pour stopper les threads d'un agent qui souhaite migrer ou mourir. Notre implémentation suggère d'utiliser la notion d'agent invalidée mentionnée plus haut (3.1.3). Un agent invalidée ne peut migrer, et ne peut participer à aucune interaction avec un quelconque objet du réseau<sup>2</sup>. Par contre, il est possible d'utiliser une copie. Lorsqu'un serveur fait migrer un agent, l'objet restant est marqué invalide. Les threads qui opèrent sur l'agent peuvent donc tester cet état grâce à la méthode `isValid()` de la classe `Agent`. Malgré tout, le serveur doit faire "confiance" aux threads pour s'arrêter. Il est en effet trivial de surcharger un serveur en effectuant une boucle infinie dans une thread.

Cette notion de "confiance" n'est pas convenable et une esquisse de solution est donnée en 3.3.9.

### 3.2.3 Références et copies

L'un des problèmes majeurs auquel nous avons été confronté est le problème lié à la notion de référence. En effet, supposons qu'un objet créé un agent sur un serveur du réseau, puis souhaite communiquer avec cet objet (une application triviale et nécessaire est l'interface pilote du serveur). Elle doit donc obtenir une référence de cet objet. Cette notion de référence ne pose pas de problèmes dans les applications non distribuées classiques. En effet, au sein d'une même JVM, les références sont des pointeurs. Cependant, l'utilisation du protocole RMI pour notre système d'agents modifie cette notion de référence en distinguant trois cas :

- si l'objet référencé est un objet RMI, alors la référence peut-être considérée comme un pointeur (distant) d'objet;
- si l'objet référencé est un objet non RMI renvoyé par une méthode d'un objet RMI, ou passé en paramètre à une méthode d'un objet RMI, alors l'objet - qui doit être **Serializable** - est **copié**;
- dans les autres cas, la notion de référence habituelle est valide.

Toute instance de notre classe serveur `AgentServer` étant un objet RMI, ses méthodes font des copies de ses arguments et renvoient des copies en valeur de retour. Nous aurions pu par exemple avoir une fonction `createAgent` renvoyant un objet de la classe `Agent`. Mais en procédant de la sorte, nous aurions eu dans notre réseau, deux agents, l'un dans le serveur et l'autre dans la JVM de l'objet appelant. Toutes les méthodes de la classe `AgentServer` posent ce problème.

Une solution est de renvoyer une référence distante de l'agent. C'est la notion de *proxy* des Aglets d'IBM. On accède à l'agent via son proxy. Lui seul est capable de communiquer directement avec l'agent. Malgré tout, cette notion de proxy est assez lourde à gérer. En effet, une fois obtenue le proxy d'un agent, si l'agent migre, que devient le proxy? Intuitivement, il devrait rester valide et connaître la destination de son agent. Il y a plusieurs solutions à ce problème (par exemple l'agent ou le serveur doit broadcaster aux proxy<sup>3</sup> la destination à chaque migration). Ces solutions peuvent devenir très lourdes (et très lente) s'il y a beaucoup d'agents migrants et beaucoup de références. De plus, lorsque la migration échoue, que deviennent les proxy? Cette solution n'a donc pas été retenue.

La solution retenue utilise la notion de référence indirecte d'un agent. La différence entre une référence distante (proxy) et une référence indirecte est que cette dernière ne permet pas la manipulation de l'agent directement comme le permet un proxy. Par exemple, un proxy possède la méthode `migrate` qui fait référence à la méthode de même nom de son agent associé. Dans notre implémentation, la référence indirecte est un *id*<sup>4</sup>. C'est par l'intermédiaire de cet *id* que l'on peut communiquer avec l'agent via la méthode `callAgentMethod`.

Ainsi, nous assurons l'unicité d'un agent à travers le réseau. De plus, grâce à la pseudo-validité référentielle, si l'agent migre, sa référence sur le serveur de départ est invalidée jusqu'à la mort de l'agent. Si l'agent revient, la référence est à nouveau valide.

---

2. Sauf les objets qu'il a lui-même créés et qui n'utilisent pas les fonctionnalités du système d'agents (méthodes incontrôlables entre autres)

3. Plusieurs objets peuvent vouloir une référence sur l'agent ce qui nécessite autant de proxy que de références.

4. Nombre entier sous forme de chaîne de caractères

Enfin, pour faciliter l'écriture des applications, et pour éviter trop de messages inutiles sur le réseau, il est possible d'obtenir une copie d'un agent par appel à la méthode `getAgentCopy`. Cependant, cette copie n'est pas un agent. Elle représente l'état d'un agent à un instant donné. Les actions effectuées sur cette copie n'ont aucun effet sur l'agent. Cela permet par exemple, de récupérer les données calculées par un agent à un instant donné, et de les représenter sans le perturber.

### 3.2.4 Tolérance de panne

L'implémentation de la tolérance de panne ne pose pas de problème particulier (nous avons vu en 3.1.4 qu'il suffisait d'utiliser comme référence d'un serveur son adresse au format URL). Cependant, c'est l'action de tolérance qui remet en cause ce que nous avons appelé la pseudo-validité référentielle.

En effet, nous avons vu qu'un serveur établit une correspondance entre agent et *id*. Cela permet, lorsqu'un agent revient sur un serveur, de lui donner le même *id* que lors de sa première visite. La référence d'un objet sur l'agent migrateur avait été invalidée, n'autorisant plus son interaction avec l'image invalidée (rappelons qu'il s'agit de l'objet restant une fois l'agent migré). L'agent étant revenu, la référence est à nouveau valide, et les communications entre l'objet et l'agent peuvent continuer.

Malheureusement, cette propriété ne peut-être garantie si la tolérance de panne est gérée. En effet, si un serveur tombé en panne est redémarré, sa table de correspondance est remise à zéro, les agents l'ayant visité ne peuvent donc retrouver leurs *ids* originaux. La pseudo-validité référentielle n'est donc plus satisfaite.

Une proposition de solution sera donnée en 3.3.8

### 3.2.5 Sécurité

Les agents posent des problèmes de sécurité qui, s'ils ne sont pas résolus, ne permettent pas le déploiement de serveurs d'agents sur Internet. Il est en effet trivial de créer des "virus" ou des "vers" avec des agents. Notamment, notre implémentation ne se soucie pas des ressources utilisées par les agents. Un agent peut donc écrire sur le disque, accéder au réseau, utiliser la mémoire et la CPU sans aucune restriction autre que celles du système d'exploitation. Pire, un serveur accepte tout agent sans se soucier de son origine. Il peut devenir nécessaire de trouver une solution si l'on souhaite utiliser un serveur d'agents accessibles à tous.

Une solution sera donné en 3.3.9.

## 3.3 Évolutions possibles

Cette section est dédié à l'évolution du serveur que nous avons implémenté. Chaque sous-section donne soit une idée de solution pour résoudre un problème évoqué dans la section précédente, soit une évolution du serveur pour le rendre plus rapide ou pour lui ajouter des fonctionnalités.

### 3.3.1 Recherche d'agents

Il est tout à fait possible dans notre implémentation de rechercher un agent avec un *id* donné. En effet, en plus de la correspondance agent-*id* qu'établissent les serveurs (3.1.3), l'image invalidée contient l'*id* que l'agent a reçu en arrivant sur le serveur destinataire de sa migration. Par conséquent, pour un *id* local à un serveur donné, il est facile pour le serveur exécutant la requête de rechercher l'agent sur le réseau. Il suffit d'utiliser l'algorithme suivant:

- Si *id* est libre, l'agent est mort.
- Si *id* est validé, l'agent est trouvé sur ce serveur.
- Sinon, utiliser dans l'image invalidée, le serveur sur lequel a été l'agent lors de sa migration. Poser la question au serveur trouvé.

Ainsi, de serveur en serveur, l'agent finira par être trouvé. Il deviendra même possible de simuler la notion de proxy vu en 3.2.3<sup>5</sup>.

---

5. Ce qui ne résoudra pas les problèmes liés à cette notion

### 3.3.2 Broadcast de la mort d'un agent

Le broadcast de la mort d'un agent n'a pas été implémenté efficacement puisqu'il utilise  $n$  étapes où  $n$  est le nombre de serveurs à avertir. Il serait plus judicieux d'utiliser l'algorithme classique en  $\log_2(n)$  étapes schématisé sur la figure 3.3.2.

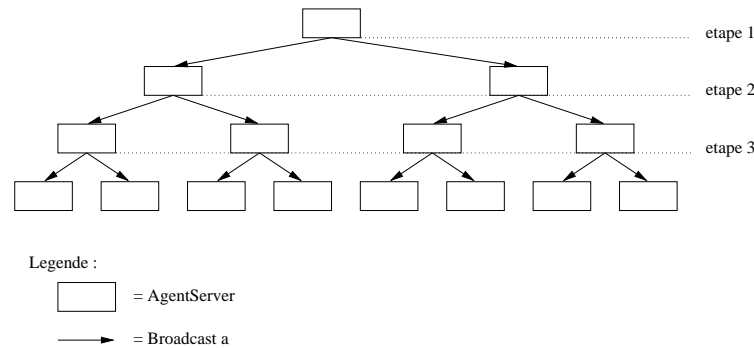


FIG. 3.3: Schéma du broadcast en  $\log_2(n)$  étapes

### 3.3.3 Chargement de code à distance

La méthode `createAgent` attend en paramètre une URL représentant l'adresse de la classe à charger. Si cette classe n'est pas locale au serveur, une exception est renvoyée. Nous pourrions autoriser le chargement d'une classe externe. Rien n'interdit en effet de charger une classe qui soit sur un serveur quelconque, puis de l'instancier en un agent. Il suffit d'implémenter une classe `NetClassLoader` qui étende la classe `java.lang.ClassLoader`. Le lecteur pourra se référer à l'API de Sun pour de plus amples détails.

Dans le cas où la classe est accessible par un serveur d'agent distants, nous pourrions nous demander s'il n'est pas équivalent de demander à l'agent distant de créer l'agent puis de l'envoyer au serveur qui a fait la requête comme schématisé sur la figure 3.3.3.

Il est clair qu'il n'y a pas équivalence. L'une des raisons concerne les vœux de l'agent : un agent peut souhaiter être non-migrable et le serveur doit respecter ce souhait. Par ailleurs, cette possibilité pose des problèmes conceptuels : l'agent doit-il savoir qu'il a été migré à son insu ? Qui est son serveur d'origine ? Pour toutes ces raisons, et surtout parce que de toutes façon, cette solution ne permet pas de charger du code distant depuis un serveur qui n'est pas un serveur d'agent, il est plus raisonnable d'implémenter la première solution i.e un `NetClassLoader`.

### 3.3.4 Désactivation d'un agent

La section 3.2.1 soulève - entre autres - le problème de la désactivation d'un agent soit parce qu'il migre, soit parce qu'il souhaite mourir. Cette désactivation est problématique puisqu'elle invoque l'appel d'une méthode incontrôlables (`onMigrating` et `onDisposing`). L'allocation d'une thread pour cette méthode est paradoxale avec l'idée de terminaison. Pourtant, nous l'avons vu, cela est *nécessaire* pour toutes les méthodes incontrôlables (section 3.2.1).

Une solution envisageable est de donner un délai à chacune de ces méthodes avant de marquer l'agent invalide. Ce délai pourrait même être passé en paramètre pour permettre à l'agent de choisir sa désactivation en fonction du temps qu'il lui reste. Nous verrons dans que cette solution s'accorde avec celle donnée dans la section (3.3.5).

### 3.3.5 Mort du serveur

L'appel de la méthode `exit` d'un serveur entraîne l'appel de la méthode `dispose` de tous ses agents. Or, nous l'avons vu, cette dernière entraîne l'appel à la méthode `onDisposing` qui est incontrôlable ! L'une de ces méthodes peut rentrer dans une boucle infinie ou bien dans une étape de calcul très long... Il n'est pas possible de différencier les deux cas. Aussi, une solution est d'attendre un certain délai. Si ce délai est dépassé par une méthode `dispose` alors s'il reste des méthodes à appeler, on les invoque, sinon, on termine le serveur sans se préoccuper des threads qui continue de tourner.

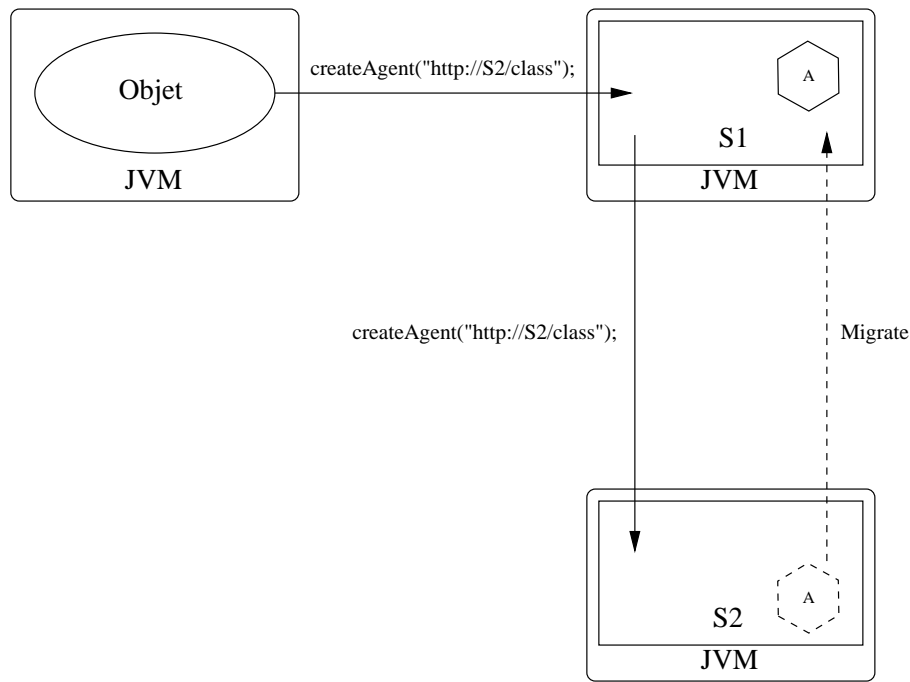


FIG. 3.4: Simulation de chargement du code d'un agent à distance ?

Nous pourrions aussi créer une méthode de dernier recours `lastResort` dans la classe `Agent` qui serait appelé par le serveur après invocation de sa méthode `exit`. Un délai pourrait même être passé en argument à cette méthode. Ainsi, un serveur très chargé pourrait laisser un délai plus long. Enfin, il est aussi possible de fournir en argument à `lastResort` le nom d'un serveur de secours sur lequel les agents pourraient se réfugier pour continuer leur travaux.

### 3.3.6 MobilityListener ...

Les méthodes incontrôlables peuvent être dynamiques. Un agent peut avoir une méthode `onMigrating` sur un serveur, puis en changer. Une implémentation possible est d'utiliser la notion de *listener*. Un agent possède une liste d'actions à effectuer lorsqu'il va migrer, a migré, va mourir... On peut ajouter des actions à cette liste ou en retirer dynamiquement. Un serveur au lieu d'appeler la méthode `onMigrating` par exemple, invoque toutes les actions de la liste associées à l'état "va migrer".

### 3.3.7 Interaction dynamique et passage de message

La sous-section 3.2.1 soulève entre autres le problème de l'interaction d'un agent avec d'autres objets. En effet, n'importe quel objet ayant une référence indirecte sur un agent peut appeler une de ses méthodes publiques et notamment les méthodes incontrôlables.

Heureusement, pour appeler la méthode d'un agent, il faut passer par son serveur local via la méthode `callAgentMethod`. Le serveur peut donc vérifier qu'aucune méthode incontrôlable n'est invoquée par un autre objet que lui. Par contre, l'agent peut implémenter plusieurs méthodes publiques inconnues du serveur sans souhaiter leur appel par n'importe quel objet. Si le serveur doit garantir que seuls les objets autorisés par l'agent peuvent appeler une de ses méthodes, une table de correspondance clés privées - actions autorisées peut-être utilisée. L'appel à la méthode `callAgentMethod` nécessiterait une structure `Ticket` qui aura été fournie par le serveur après un appel à une méthode `getTicketForMethod`. Cette dernière prendrait en paramètre une clé et renverrait un ticket pour l'appel d'une méthode.

Une évolution intéressante de notre système d'agents et la gestion d'une messagerie. Considérons un agent statique - i.e non migrable - dont le seul but est de transmettre des messages aux agents du serveur. Le passage de message point à point, synchrone, asynchrone, multicast et broadcast serait géré par cet agent. Un agent enverrait un message du type "je m'inscris au groupe de message de type HTTP". Dès lors, tous message de type "HTTP" serait envoyé à

tous les agents membres du groupe. Nous aurions donc le même système que les aglets d'IBM mais implémentée dans une couche au dessus du serveur.

### 3.3.8 Tolérance de panne

Nous avons vu que la tolérance de panne entraîne le non respect de la pseudo-validité référentielle. Une solution à ce problème pourrait être de sauvegarder sur disque la table de correspondance afin d'en retrouver au moins une partie lors de la remise en route. Malgré tout, la libération des ids des agents qui meurent sur le réseau n'étant pas possible lorsque le serveur est en panne, la table de correspondance sur disque ne peut être mise à jour. Même si cette dernière conséquence n'est pas dramatique, il deviendrait nécessaire d'utiliser un ramasse-miettes pour éviter la profusion d'*ids* inutiles. Un tel ramasse-miettes peut-être implémenté en utilisant la recherche d'agents présentée dans la section 3.3.1.

### 3.3.9 Sécurité

La notion de sécurité devient nécessaire si un serveur doit accepter des agents dont il ne connaît pas l'origine. Ce point a été vu en 3.2.5. Pour limiter l'accès au ressources du système, il est recommandé d'utiliser la classe `java.lang.SecurityManager` et `java.rmi.RMISecurityManager`. Le lecteur désireux de faire évoluer notre serveur en une version qui gère mieux la sécurité se reportera à l'API du jdk. Le langage Sumatra [1] peut-être intéressant puisqu'il a été développé pour combler les lacunes de Java en matière d'allocations de ressources pour les agents mobiles. Il est, de plus, entièrement compatible avec Java puisqu'il en reprend la JVM.

La formalisation de cette implémentation sera vu dans le chapitre 6.

# Chapitre 4

## Choix d'un formalisme

Maintenant que nous avons implémenté notre modèle, il convient de le décrire dans un formalisme qui permette de prouver des propriétés sur les agents. Se pose donc le problème du choix du formalisme. Il existe en effet plusieurs modèles de calculs de processus concurrents, dont le précurseur est CCS (Calculus of Concurrent Processes)[21].

Sans entrer dans les détails, CCS ne permet pas de formaliser simplement notre modèle. Sa puissance d'expression ne nous suffit pas. Aussi, nous avons recherché un modèle de calcul adéquat. Plusieurs ont été trouvés. Certains ne nous ont pas été utiles mais sont mentionnés à titre indicatif. D'autres par contre ont été découverts à un stade trop avancé de notre travail pour les appliquer à notre modèle.

### 4.1 Le $\pi$ -calcul

Ce modèle de calcul ([38]) proposé par Milner, Parrow et Walker vise à améliorer le pouvoir d'expression de CCS en autorisant le passage de "canaux" entre processus. Une première extension de CCS nommée ECCS avait été présentée par Engberg et Nielsen [45]. Cependant, le  $\pi$ -calcul simplifie et généralise ses prédécesseurs. Depuis, beaucoup de travaux portent sur le  $\pi$ -calcul: ils visent soit à améliorer son pouvoir d'expression, soit à mieux comprendre la théorie sous-jacente. Les travaux de D. Sangiorgi en particulier nous ont permis de mieux comprendre le  $\pi$ -calcul.

#### 4.1.1 L'étude de D. Sangiorgi

Cette étude [39] permet de mieux assimiler les difficultés du  $\pi$ -calcul (principalement la dissymétrie entre l'envoi et la réception d'un nom). D. Sangiorgi définit une hiérarchie de calculs de premier ordre basés sur la notion de mobilité interne. Leur pouvoir d'expression est croissant mais fini. Le plus expressif d'entre eux - noté  $\pi I$  - semble être aussi expressif que le  $\pi$ -calcul tout en étant théoriquement plus simple. Il montre en outre que l'ajout de l'ordre supérieur dans le  $\pi$ -calcul n'augmente pas son pouvoir d'expression.

#### 4.1.2 Polymorphisme

B.Pierce et D.Sangiorgi proposent ([25]) une extension du  $\pi$ -calcul qui gère le polymorphisme. L'idée est d'autoriser l'envoi d'un type en plus de l'envoi d'un nom. Ainsi, il est possible d'écrire des expressions qui reçoivent une valeur sans en connaître le type.

#### 4.1.3 Raffinement du typage

Plusieurs études portent sur le typage du  $\pi$ -calcul.

Par exemple, B.Pierce et D.Sangiorgi étendent dans [24], la syntaxe du  $\pi$ -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.

Une autre étude intéressante concernant le typage du  $\pi$ -calcul est présentée par B.Pierce, N.Kobayashi et D.Turner dans [17]. 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 certain nombre de fois dans le sens correspondant à sa polarité mais pas plus.

#### 4.1.4 Asynchronisme

La recherche de G.Boudol [5] a pour but de définir le  $\pi$ -calcul à partir d'un modèle de calcul de plus bas niveau (une machine chimique abstraite cf 4.4) dans lequel le passage de messages s'effectue de manière asynchrone. Le passage de messages synchrone pouvant être simulé par le passage de messages asynchrone à l'aide d'un signal d'accusé de réception, le  $\pi$ -calcul peut-être codé dans ce nouveau modèle de calcul. Cette étude diffère de celle effectuée par K.Honda et M.Tokoro [15] dans le sens où elle n'utilise pas de notion de bisimulation mais une adaptation de l'observabilité des  $\lambda$ -termes.

#### 4.1.5 Outils

Nous l'avons vu, le  $\pi$ -calcul est l'objet de nombreuses recherches. En outre, plusieurs outils sont en cours de développement pour faciliter son étude et l'étude de modèles écrits dans ce formalisme. Citons les travaux de T.Melham [42] qui donne une méthode pour axiomatiser le  $\pi$ -calcul dans l'assistant à la démonstration HOL [41].

Le  $\pi$ -calcul peut-être comparé au  $\lambda$ -calcul inventé par Church [9]. 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  $\pi$ -calcul sont à l'origine de familles de langage. CCS est associé au langage Facile [44]. Le  $\pi$ -calcul lui est implémenté dans le langage Pict. Ce dernier propose certaines extensions déjà mentionnées (polymorphisme et sous-typage) mais en est encore à un stade expérimental.

Nous n'avons pas trouvé autant de documentation sur un autre langage, autant de recherche et d'outils. Par ailleurs, le  $\pi$ -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 6.1.1.

## 4.2 CHOCS

CHOCS [43] (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  $\pi$ -calcul(cf [39] section 7.3 p26).

## 4.3 Le spi-calcul

Le spi-calcul [12] étend le  $\pi$ -calcul avec des primitives de cryptographie. Il n'y a aucun intérêt à l'utiliser pour notre travail.

## 4.4 La Machine chimique abstraite

La machine chimique abstraite [4] n'est pas un formalisme à proprement parler. Il propose une syntaxe et une sémantique très différentes de CCS et de ses dérivées. Il utilise la notion de molécules et de réactions chimiques afin de donner une structure sémantique utilisable pour définir (ou coder) de nouveaux calculs.

## 4.5 Le join-calcul

Le join-calcul [13] est une reformulation du  $\pi$ -calcul avec une machine chimique abstraite. La notion de place d'interaction est mieux définie. Ce formalisme a donné naissance au langage de même nom. Des dérivés de ce calcul existent et peuvent être intéressants pour notre modélisation (notamment le join-calcul distribué). De plus, le join-calcul est strictement équivalent au  $\pi$ -calcul. L'un est codable en l'autre et vice-versa. Malheureusement, faute de temps, nous n'avons pu étudier une modélisation des agents mobiles en Java dans ce formalisme.

## 4.6 Les Ambiances mobiles

Le calcul basé sur la notion d'ambiances mobiles [7] est intéressante. Il fournit une puissance d'expression suffisante pour coder le  $\pi$ -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 *à priori* plus simple de modéliser un système d'agents avec ce système. Cependant, trop récemment découvert, nous n'avons pas pu modéliser les agents mobiles dans ce formalisme.

## 4.7 Conclusion : choix du $\pi$ -calcul

Le  $\pi$ -calcul a été choisi principalement pour les raisons suivantes:

- il permet une modélisation assez proche de notre implémentation;
- dérivé de CCS, qui est utilisé depuis longtemps, il en reprend les principes syntaxiques facilitant son apprentissage;
- il est l'objet de beaucoup d'études pour le rendre plus expressif, nous en avons vu quelques-unes;
- une abondante littérature est disponible facilement ce qui est un atout non négligeable;
- plusieurs outils sont développés pour étudier les modèles écrits en  $\pi$ -calcul ou pour créer un langage de programmation inspiré du  $\pi$ -calcul.

Le prochain chapitre est donc une introduction à ce modèle de calcul.

# Chapitre 5

## Présentation du $\pi$ -calcul

Le  $\pi$ -calcul est un modèle de calcul de processus concurrent basé sur la notion de *nommage*. Il a été présenté dans sa version d'origine dans [38] sous sa forme monadique. Puis, Robin Milner l'a étendu à une version polyadique justifiée par la notion de *sorte* (typage) qui résout les problèmes liés à l'encodage de la version polyadique en monadique. De plus, dans cette nouvelle version, il simplifie la sémantique en définissant séparément les règles de réduction et le système de transition étiquetées appelé *engagement*. C'est cette deuxième version du  $\pi$ -calcul, plus complète et plus précise que la première, que nous présentons ici.

### 5.1 Introduction

Le  $\pi$ -calcul est le fruit d'un travail visant à améliorer le pouvoir d'expression de CCS<sup>1</sup>. Il est l'extension logique des travaux de Engberg et Nielsen (cf [45]). Aussi, il existe une grande ressemblance syntaxique entre ces deux modèles de calculs. Toutefois, le  $\pi$ -calcul est d'une complexité mathématique bien plus grande que CCS. Il lui apporte essentiellement deux nouveautés:

- la suppression des notions de variables, de valeurs et de noms de canaux au profit d'une seule la 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  $\pi$ -calcul.

### 5.2 Le $\pi$ -calcul monadique

#### 5.2.1 Syntaxe

Comme nous l'avons vu, la plus petite entité primitive dans le  $\pi$ -calcul est le nom. Une autre entité est le processus.

**Définition 5.2.1 (Entités du  $\pi$ -Calcul monadique)**

Soit  $\mathcal{X} = \{x, y, \dots\}$  l'ensemble fini de noms. Les noms ne sont pas structurés. Soit  $\mathcal{Q} = \{P, Q, \dots\}$  l'ensemble des processus.

Si  $I$  est un ensemble fini d'index, les processus sont construits selon la syntaxe:

**Règles syntaxiques 5.2.1 ( $\pi$ -calcul monadique d'ordre inférieur simple)**

$$P ::= \sum_{i \in I} \pi_i . P_i \mid P \mid Q \mid !P \mid (\nu x)P$$

**Notation 5.2.1 (Processus inactif)**

Si  $I = \emptyset$ , la somme est noté  $\mathbf{0}$ .

---

1. Le lecteur n'ayant aucune connaissance de CCS pourra se référer à [21].

Dans l'expression  $\pi.P$ ,  $\pi$  est une *action atomique* : c'est la première action effectuée par  $P$ . Il existe deux formes de préfixes :

**Définition 5.2.2 (Sujets et objets d'une action)**

Si  $\pi = x(y)$ ,  $x$  est un sujet positif.

Si  $\pi = \bar{x}y$ ,  $\bar{x}$  est un sujet négatif. Dans les deux cas,  $y$  est l'objet de l'action. On appellera aussi  $\bar{x}$  le co-nom de  $x$  et l'on a  $\overline{\bar{x}} = x$ . 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  $x$ ”;
- $\bar{x}y$  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 alternatives de communications. Le processus ne peut choisir; il ne peut s'engager dans une alternative avant qu'elle n'ai lieu, et cette alternative empêche tout engagement dans une autre. Les processus sous cette forme sont appelés *processus normaux*. En effet, Robin Milner démontre dans [22] que tous processus peut-être converti sous cette *forme normale*.

**Définition 5.2.3 (Processus normaux)**

L'ensemble des processus normaux est  $\mathcal{N} = \{M, N, \dots\}$ .

Leur syntaxe est la suivante:

**Règles syntaxiques 5.2.2 (Processus normaux)**

$$N ::= \pi.P \mid \mathbf{0} \mid M + N$$

L'expression  $P|Q$  - “P parallèle Q” - symbolise une activité concurrente des processus  $P$  et  $Q$ . Ils peuvent agir indépendamment l'un de l'autre (par des communications internes) mais peuvent aussi communiquer.

**Définition 5.2.4 (Réplication)**

$$!P \stackrel{\text{def}}{=} P|!P$$

L'expression  $!P$  - “réplique  $P$ ” - est définie ainsi:

$$!P = P|!P$$

Il n'y a pas de risque de concurrence d'activité comme nous le verrons par la suite. L'opérateur “!” est appelé *réplicateur*. Une des utilisations courantes est l'expression  $!\pi.P$  : l'accès à une ressource illimitée nécessite une communication via  $\pi$ .

Enfin, l'expression  $(\nu x)P$  - “nouveau  $x$  dans  $P$ ” - restreint le nom  $x$  à  $P$ .  $x$  devient donc connu exclusivement de  $P$ . L'opérateur  $\nu$  ne correspond pas tout à fait à l'opérateur de restriction de CCS. En effet, cet opérateur crée un nouveau nom. Et de part 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  $\nu$ . Par exemple, si on a

$$P \equiv (\nu y) \bar{y}x.P' \text{ et } Q \equiv x(z).Q'$$

, 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 5.2.5 (Processus inactif)**

Un processus est inactif s'il ne peut communiquer avec aucun autre processus.

**Exemple 5.2.1 (Processus inactif)**

Les processus  $(\nu x)(\nu z)\bar{x}z$  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  $x(y).0$  et  $\bar{x}y.0$  sont si courants que nous omettrons le  $.0$  final et écrivons simplement  $x(y)$  et  $\bar{x}y$ . Donc:

**Notation 5.2.2**

$$P.0 = P$$

**5.2.2 Congruence structurelle**

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 5.2.6 (Noms libres, noms liés et ensemble de noms)**

$$\begin{aligned} bn(x(y)) &= \{y\} & , & & fn(x(y)) &= \{x\} \\ bn(\bar{x}y) &= \emptyset & , & & fn(\bar{x}y) &= \{x, y\} \\ n(P) &\stackrel{\text{def}}{=} & & & bn(P) \cup &fn(P) \end{aligned}$$

Rappelons quelques définitions :

**Définition 5.2.7 (Monoïde)**

Soit  $E$  un ensemble,  $*$  une opération associative et  $e$  un élément neutre pour  $*$ . Alors  $(E, *, e)$  est un monoïde. Si la loi  $*$  est commutative, le monoïde est dit commutatif.

Robin Milner définit alors la *congruence structurelle* - noté  $\equiv$  - ainsi:

**Définition 5.2.8 (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.  $(\mathcal{N} / \equiv, +, 0)$  est un monoïde commutatif;
3.  $(\mathcal{P} / \equiv, |, 0)$  est un monoïde commutatif;
4.  $!P \equiv P|!P$ ;
5.  $(\nu x)0 \equiv 0$ ,  $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$ ;
6. Si  $x \notin fn(P)$  alors  $(\nu x)(P|Q) \equiv P|(\nu x)Q$ .

La congruence structurelle implique les propriétés suivantes :

1. Si  $x \notin fn(P)$ ,  $(\nu x)P \equiv P$ ;
2. Toutes restrictions peut-être factorisée dans un processus qui n'est pas en forme normales.

Donnons la preuve de la première propriété :

### Preuve 5.2.1 (Propriété 1)

$$\begin{aligned}
 (\nu x)P &\stackrel{(3)}{\equiv} (\nu x)(P|\mathbf{0}) \\
 &\stackrel{(6)}{\equiv} P|(\nu x)\mathbf{0} \\
 &\stackrel{(5)}{\equiv} P|\mathbf{0} \\
 &\stackrel{(3)}{\equiv} P
 \end{aligned}$$

Pour la deuxième propriété, donnons un exemple :

### Exemple 5.2.2 (Factorisation des restrictions)

Si  $P \equiv (\nu y)\bar{x}y$ , alors

$$\begin{aligned}
 x(z).\bar{y}z|!P &\stackrel{(4)}{\equiv} x(z).\bar{y}z|(\nu y)\bar{x}y|!P \\
 &\stackrel{(1)}{\equiv} x(z).\bar{y}z|(\nu y')\bar{x}y'|!P \\
 &\stackrel{(6)}{\equiv} (\nu y')(x(z).\bar{y}z|\bar{x}y')|!P
 \end{aligned}$$

qui après communication donne

$$(\nu y')(\bar{y}'z|\mathbf{0})|!P$$

Des écritures comme

$$A(x) \stackrel{\text{def}}{=} x(y).B(y) ; \quad B(y) \stackrel{\text{def}}{=} \bar{y}z.A(z)$$

sont possibles. Milner montre en effet qu'il n'est pas nécessaire de rajouter des constructions syntaxiques au  $\pi$ -calcul: il présente un codage de la récursion en réplication que nous ne présenterons pas ici.

### 5.2.3 Règles de réduction

Avant d'aller plus loin, définissons la notion de substitution :

#### Définition 5.2.9 (Substitution)

On note  $\sigma = z/y$  une substitution : 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 normales. La première règle de réduction est la *communication* :

$$\text{COMM} : (\dots + x(y).P)|(\dots + \bar{x}z.Q) \rightarrow P\{z/y\}|Q$$

La communication survient donc entre deux processus normaux atomiques  $\pi.P$  nécessairement complémentaires (dont les sujets sont complémentaires). De plus, cette communication annihile toute autre alternative. COMM est la seule règle axiomatique axiome pour  $\rightarrow$ . Les autres sont des règles d'inférence.

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

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 trouvent ci-dessous :

$$\begin{aligned} \text{COMM} : & (\cdots + x(y).P) | (\cdots + \bar{x}z.Q) \\ & \rightarrow P\{z/y\} | Q \\ \text{PAR} : & \frac{P \rightarrow P'}{P | Q \rightarrow P' | Q} \\ \text{RES} : & \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'} \\ \text{STRUCT} : & \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'} \end{aligned}$$

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 réplique.

### Exemple 5.2.3 (Réductions non permises)

$$u(v).(x(y)|\bar{x}z) \not\rightarrow$$

car le processus  $(x(y)|\bar{x}z)$  est préfixé (nous emploierons le terme *gardé* en 5.2.10). Il doit recevoir sur le canal  $u$  avant de continuer.

$$\begin{aligned} & \text{Si } P \rightarrow P' \text{ alors } !P \not\rightarrow !P' \text{ mais} \\ & !P \equiv \underbrace{P | P | \dots | P}_n | !P \xrightarrow{n} \underbrace{P' | P' | \dots | P'}_n | !P \end{aligned}$$

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 transition 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  $\alpha$  - sujet positif ou négatif - de ceux qui ne peuvent pas. Aussi, quelques définitions sont ajoutées.

### Définition 5.2.10

Un processus  $Q$  est non gardé dans  $P$  si il intervient dans  $P$  non préfixé.

$P$  est observable à  $\alpha$  - ce que l'on écrit  $P \downarrow_\alpha$  - si  $\pi.R$  survient dans  $P$  non gardé, où  $\alpha$  est le sujet de  $\pi$  et est non restreint (i.e, on n'a pas une expression du type  $(\nu \alpha)\pi.R$ ).

### Exemple 5.2.4

$Q$  est non gardé dans  $Q | R$  et  $(\nu x)Q$  mais est gardé dans  $x(y).Q$ .

$x(y) \downarrow_x$  et  $(\nu z)\bar{x}z \downarrow_x$ , mais  $(\nu x)\bar{x}z \not\downarrow_x$  et  $(\nu x)(x(y)|\bar{x}z) \not\downarrow_x$  même si cette dernière expression a une réduction.

Une extension intéressante est le  $\pi$ -calcul polyadique qui est l'objet de la section suivante.

### 5.3 Le $\pi$ -calcul polyadique

Il est fréquent de vouloir communiquer plusieurs noms à un processus. On pourrait écrire par exemple  $\bar{x}yz$  pour envoyer le couple  $(y, z)$  sur le canal  $x$ . La réception s'écrirait  $x(ab)$ . Ces écritures seraient les abréviations naturelles respectives de  $\bar{x}y.\bar{x}z$  et  $x(y).x(z)$ . Cependant, cette abréviation donne une fausse idée des réelles communications. Considérons

$$\bar{x}y_1z_1|\bar{x}y_2z_2|x(yz)$$

par exemple le couple de noms  $(y, z)$  du processus  $x(yz)$  peut être :

$$(y, z) = (y_1, z_1) \quad (5.1)$$

$$(y, z) = (y_2, z_2) \quad (5.2)$$

$$(y, z) = (y_1, y_2) \quad (5.3)$$

$$(y, z) = (y_2, y_1) \quad (5.4)$$

Si les égalités (5.1) et (5.2) sont bien souhaitées, il n'en est pas de même des égalités (5.3) et (5.4). La solution consiste à utiliser un nom privé. Par conséquent, nous adopterons la convention<sup>2</sup> suivante :

#### Notation 5.3.1 (Codage du $\pi$ -calcul polyadique en $\pi$ -calcul monadique)

$$\begin{aligned} x(y_1 \cdots y_n) &= x(w).w(y_1).\cdots.w(y_n) \\ \bar{x}y_1 \cdots y_n &= (\nu w)\bar{x}w.\bar{w}y_1.\cdots.\bar{w}y_n \end{aligned}$$

Si  $n = 0$ ,  $x = x()$ .

Ainsi, dans l'exemple précédent, si on suppose que c'est le premier processus qui communique, on a :

$$\begin{aligned} P \equiv \bar{x}y_1z_1|\bar{x}y_2z_2|x(yz) &= (\nu w)\bar{x}w.\bar{w}y_1.\bar{w}z_1|(\nu w)\bar{x}w.\bar{w}y_2.\bar{w}z_2|x(w).w(y).w(z) \\ &\stackrel{1}{\equiv} (\nu w_1)\bar{x}w_1.\bar{w}_1y_1.\bar{w}_1z_1|(\nu w_2)\bar{x}w_2.\bar{w}_2y_2.\bar{w}_2z_2|x(w).w(y).w(z) \\ &\rightarrow \mathbf{0}|(\nu w_2)\bar{x}w_2.\bar{w}_2y_2.\bar{w}_2z_2|\mathbf{0} \equiv Q \end{aligned}$$

Et donc en utilisant la règle STRUCT, on a bien

$$P \rightarrow Q$$

La communication supplémentaire induite ne pose donc aucun problème.

Toutefois, Milner propose de rendre les communications polyadiques primitives. La raison est que la notation 5.3.1 permet des écritures du genre  $\bar{x}y_1 \cdots y_n|x(z_1 \cdots z_l)$  et ne précise pas si  $n < l$  ou si  $n > l$ . La notion de sorte - un typage un peu particulier que nous verrons en 5.4 - résoudra cette incohérence. Pour cela, il introduit la notion d'*abstractions* dans le  $\pi$ -calcul.

#### Définition 5.3.1 (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  $\lambda$ -calcul, car ici, les noms liés ne peuvent être instanciés qu'en des noms, jamais en des termes composés. Nous écrivons désormais

#### Notation 5.3.2

$$\begin{aligned} K(x_1, \dots, x_n) &\stackrel{\text{def}}{=} P \iff K \stackrel{\text{def}}{=} (\lambda x_1 \cdots x_n)P \\ x(y).P &\stackrel{\text{def}}{=} x.(\lambda y).P \end{aligned}$$

2. Momentanément comme nous le verrons en 5.3.

Nous dirons que, dans le processus  $x.(\lambda y).P$ ,  $x$  est la *localisation* de l'abstraction  $(\lambda y).P$  ainsi que dans le processus  $x.(\lambda y_1 \cdots y_n).P$ . 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 5.3.2 (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 5.3.3**

$$\bar{x}y_1 \cdots y_n.P \stackrel{\text{def}}{=} \bar{x}.[y_1 \cdots y_n]P$$

Dans le processus  $\bar{x}.[y_1 \cdots y_n]P$ ,  $\bar{x}$  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.

On introduit quelques notations seront les bienvenues :

**Notation 5.3.4**

$$\begin{aligned} \bar{x}.[\bar{y}].\mathbf{0} &= \bar{x}.[\bar{y}] \\ \bar{x}.[]P &= \bar{x}.P \\ \bar{x}.[]\mathbf{0} &= \bar{x} \end{aligned}$$

D'où la nouvelle syntaxe suivante :

**Règles syntaxiques 5.3.1 ( $\pi$ -calcul polyadique)**

$$\begin{aligned} \text{Processus Normaux : } N & ::= \alpha.A \mid \mathbf{0} \mid M + N \\ \text{Processus : } P & ::= N \mid P \mid Q \mid !P \mid (\nu x)P \\ \text{Abstractions : } F & ::= P \mid (\lambda x)F \mid (\nu x)F \\ \text{Concrétions : } C & ::= P \mid [x]C \mid (\nu x)C \\ \text{Agents : } A & ::= F \mid C \end{aligned}$$

Nous devons redéfinir la relation de congruence structurelle vue en 5.2.8 pour qu'elle prenne en compte la nouvelle syntaxe. En réalité, nous gardons les règles 1 à 6 et nous y ajoutons trois autres règles.

**Définition 5.3.3 (Congruence structurelle  $\equiv$  du  $\pi$ -calcul polyadique)**

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 monoïde symétrique;
3.  $(\mathcal{P} / \equiv, \mid, \mathbf{0})$  est un monoïde symétrique;
4.  $!P \equiv P \mid !P$ ;

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.  $(\nu y)[x]C \equiv [c](\nu y)C \quad (x \neq y)$
9.  $(\nu x)(\nu y)A \equiv (\nu y)(\nu x)A, (\nu x)(\nu x) \equiv (\nu x)$

**Définition 5.3.4 (Forme standard)**

Toute abstraction ou concrétion est convertible en une forme standard :

$$F \equiv (\lambda \vec{x})P \text{ et} \\ C \equiv (\nu \vec{y})[\vec{x}]P \text{ avec } (\vec{y} \subseteq \vec{x})$$

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 :

$$x(\vec{y}).P|\vec{x}\vec{z}.Q \rightarrow P\{\vec{z}/\vec{y}\}|Q$$

où  $\vec{y}, \vec{z}$  sont de même taille, les composantes de  $\vec{y}$  étant toutes distinctes.

**Définition 5.3.5 (Pseudo-application)**

On appelle pseudo-application la communication entre une abstraction et une concrétion :

$$F \equiv (\lambda \vec{x})P, C \equiv (\nu \vec{z})[\vec{y}]Q, \vec{x} \cap \vec{z} = \emptyset \text{ et}$$

$$w.F|\vec{w}.C \rightarrow F \bullet C \stackrel{\text{def}}{=} (\nu \vec{z})(P\{\vec{y}/\vec{x}\}|Q)$$

**Définition 5.3.6 (Relation de réduction)**

Notre relation de réduction est donc la plus petite relation satisfaisant les règles suivantes:

$$\text{COMM} : (\dots + x.F)|(\dots + \vec{x}.C) \rightarrow F \bullet C$$

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

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 défini comme suit:

**Définition 5.3.7 (Application)**

$$((\lambda x)F)y \stackrel{\text{def}}{=} (\nu w)(w.(\lambda x)F)|\vec{w}.[y] \\ \rightarrow (\nu w)(F\{y/x\} | \mathbf{0}) \\ \equiv F\{y/x\}$$

Par conséquent, toute instances d'application peut-être éliminée en utilisant la congruence structurelle.

Cependant, si l'on veut introduire des définitions récursives d'abstractions constantes, comme

$$K \stackrel{\text{def}}{=} F_K$$

où  $F_K$  est une abstraction qui peut contenir  $K$  et d'autres abstractions constantes, l'application devient une abréviation<sup>3</sup> indispensable.

### Définition 5.3.8

Soit  $P = \alpha.A$ , un processus atomique normal. Alors  $\alpha$  est une action<sup>4</sup> et  $A$  est une continuation.  $\alpha.A$  sera appelé un engagement.

Ainsi,  $P$  est un processus qui peut s'engager à  $\alpha$ . 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 à  $\alpha$ ". C'est exactement ce que le système de transition de [38] décrit, avec une notation différente. Par exemple, à la place de  $P \succ \bar{x}.[y]P'$ , la transition étiquetée  $P \xrightarrow{\bar{x}y} P'$  est utilisée. Avant de continuer, nous devons introduire l'action inobservable  $\tau$  :

### Définition 5.3.9 (Action inobservable $\tau$ )

$$\tau.P \stackrel{\text{def}}{=} (\nu x)(x.P|\bar{x}); \quad x \notin fn(P)$$

et dorénavant, les variables de noms pourront prendre la valeur  $\tau$  :

$$\alpha, \beta, \dots \in \mathcal{X} \cup \{\tau\}$$

De plus, nous définissons la syntaxe suivante :

### Règles syntaxiques 5.3.2 (Composition d'abstractions ou de concrétions)

$$F \equiv (\lambda \vec{x})P; \quad G \equiv (\lambda \vec{y})Q$$

$$\vec{x} \notin nm(G); \quad \vec{y} \notin nm(P)$$

$$F|G \stackrel{\text{def}}{=} (\lambda \vec{x}\vec{y})(P|Q)$$

De même

$$C \equiv (\nu \vec{x})[\vec{u}]P; \quad D \equiv (\nu \vec{y})[\vec{v}]Q$$

$$\vec{x} \notin D; \quad \vec{y} \notin nm(C)$$

$$C|D \stackrel{\text{def}}{=} (\nu \vec{x}\vec{y})[\vec{u}\vec{v}](P|Q)$$

$|$  est associative vis à vis de  $\equiv$ , mais pas commutative en général bien qu'elle le soit sur les processus. 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,  $A|P \equiv P|A$ . Notre relation d'engagement est définie comme suit :

### Définition 5.3.10 (Relation d'engagement)

La relation d'engagement entre processus et engagements est la plus petite relation satisfaisant les règles suivantes:

$$\text{SUM} : \dots + \alpha.A \succ \alpha.A$$

3. Abréviation car, comme nous l'avons vu en 5.2.2, la récursion peut-être éliminée par la réplication

4. En fait, le terme *localisation* d'une action serait plus juste, mais nous emploierons indifféremment l'un où l'autre.

$$\begin{aligned}
\text{COMM} &: \frac{P \succ x.F \quad Q \succ \bar{x}.C}{P|Q \succ \tau.(F \bullet C)} \\
\text{PAR} &: \frac{P \succ \alpha.A}{P|Q \succ \alpha.(A|Q)} \\
\text{RES} &: \frac{P \succ \alpha.A}{(\nu x)P \succ \alpha.(\nu x)A} (\alpha \notin \{x, \bar{x}\}) \\
\text{STRUCT} &: \frac{Q \equiv P \quad P \succ \alpha.A \quad A \equiv B}{Q \succ \alpha.B}
\end{aligned}$$

Enfin, avant de donner la définition de la bisimulation, nous introduisons une nouvelle définition:

**Définition 5.3.11 (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  $C \equiv (\nu \bar{x})[\bar{y}]P$  et  $D \equiv (\nu \bar{x})[\bar{y}]Q$  et  $C \equiv D$  alors  $P \equiv Q$ ;
- si  $F \equiv G$  alors  $|F| = |G| = n$  et  $\forall \bar{y}, |\bar{y}| = n, F\bar{y} \equiv G\bar{y}$ .

**Définition 5.3.12 (Simulation, bisimulation et bisimilarité)**

Une relation  $\equiv$  sur les agents est une simulation forte si elle est respectable et si  $P \equiv Q$  et  $P \succ \alpha.A$  alors  $\exists B/Q \succ \alpha.B$  et  $A \equiv B$ .  $\equiv$  est une bisimulation forte si  $\equiv$  et sa réciproque sont des simulations fortes. La forte bisimilarité - noté  $\sim$  est la plus large bisimulation forte.

Aussi, pour prouver que  $P \sim Q$ , 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 préservé par substitution de noms.

**Exemple 5.3.1 (Bisimilarité non préservée par substitution de noms)**

$$\bar{x}|y \sim \bar{x}.y + y.\bar{x}$$

Car,

$$\begin{aligned}
\bar{x}|y &\succ \bar{x}.(\mathbf{0}|y) \\
&\succ y.(\bar{x}|\mathbf{0}) \\
\bar{x}.y + y.\bar{x} &\succ \bar{x}.y \\
&\succ y.\bar{x}
\end{aligned}$$

Donc

$$\mathcal{R} = \equiv \cup \{(\bar{x}.(\mathbf{0}|y); \bar{x}.y)\} \cup \{(y.(\bar{x}|\mathbf{0}); y.\bar{x})\} \cup \{(\bar{x}|y; \bar{x}.y + y.\bar{x})\}$$

est une bisimulation. En effet,  $\mathcal{R}$  est respectable (évident) et on a bien

$$P \equiv \bar{x}|y \quad \mathcal{R} \quad \bar{x}.y + y.\bar{x} \equiv Q$$

par construction, et

$$(P \succ \alpha.A \Rightarrow \exists B/Q \succ \alpha.B; A \mathcal{R} B)$$

Par contre, pour  $\sigma = y/x$ ,

$$P\sigma \equiv \bar{x}|x \not\sim \bar{x}.x + x.\bar{x} \equiv Q\sigma$$

$P$  peut en effet s'engager à  $\tau.\mathbf{0}$  ce que ne peut faire  $Q$ .

Nous définissons donc :

**Définition 5.3.13 (Congruence forte)**

$P$  et  $Q$  sont fortement congruent - et l'on écrit  $P \sim Q$  - si  $P\sigma \sim Q\sigma$  pour toute substitution  $\sigma$ .

## 5.4 Sortes

Les sortes sont les équivalents en  $\pi$ -calcul du typage du  $\lambda$ -Calcul.

### Définition 5.4.1 (Sorte de sujet)

Soit  $I \subset \mathbb{N}$  et  $\mathcal{S} = \{S_i, i \in I\}$ . On appelle une sorte de sujets 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 5.4.2 (Sorte d'objet)

Une sorte d'objet est une séquence sur  $\mathcal{S}$  :

$$Ob(\mathcal{S}) = \mathcal{S}^*$$

Nous écrirons  $(S_1, \dots, S_n)$  pour une sorte d'objet. La sorte d'objet vide est  $()$ . On note  $\hat{\phantom{x}}$  l'opération de concaténation de sorte d'objets:

$$(S_1, \dots, S_n) \hat{\phantom{x}} (S'_1, \dots, S'_m) = (S_1, \dots, S_n, S'_1, \dots, S'_m)$$

### Définition 5.4.3 (Typage)

Un typage sur  $\mathcal{S}$  est une fonction partielle:

$$ob : \mathcal{S} \rightarrow Ob(\mathcal{S})$$

Si  $ob$  est finie, on écrira  $ob = \{S_1 \mapsto ob(S_1), \dots, S_n \mapsto ob(S_n)\}$ .

Un typage détermine pour tout nom  $x \in \mathcal{X}$  la sorte d'objet qu'il peut transporter.

### Définition 5.4.4 (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 :

$$\begin{array}{c} \mathbf{0} : () \quad \frac{x : S \quad F : ob(S)}{x.F : ()} \\ \frac{x : S \quad C : ob(S) \quad P : ()}{\bar{x}.C : ()} \quad \frac{}{\tau.P : ()} \\ \frac{M : () \quad N : ()}{M + N : ()} \quad \frac{P : () \quad Q : ()}{P|Q : ()} \\ \frac{P : ()}{!P : ()} \quad \frac{A : s}{(\nu x)A : s} \\ \frac{x : S \quad F : s}{(\lambda x)F : (S)\hat{s}} \quad \frac{x : S \quad C : s}{[x]C : (S)\hat{s}} \end{array}$$

Nous pourrions montrer que

Si  $x : S, y : S, A : s$  alors  $A\{y/x\} : s$

Si  $A : s, A \equiv B$ , 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  $\pi$ -calcul monadique. On peut remarquer que le codage du  $\pi$ -calcul polyadique en  $\pi$ -calcul monadique vu en 5.3.1 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  $y_i, i \in 1, \dots, n$ . C'est l'une des raisons qu'invoque Milner pour justifier l'introduction de la polyadicité.

## 5.5 Le $\pi$ -calcul d'ordre supérieur

L'idée est de permettre des écritures du genre :

$$P \stackrel{\text{def}}{=} \bar{x}.[R]P' \text{ et } Q \stackrel{\text{def}}{=} x.(\lambda X)(X|Q')$$

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  $\pi$ -calcul présenté jusqu'à présent. En effet, seul le passage de noms est admis. Toutefois, considérons le codage suivant :

$$\hat{P} \stackrel{\text{def}}{=} \bar{x}.(\nu z)[z](z.R|P') \quad \hat{Q} \stackrel{\text{def}}{=} x.(\lambda z)(\bar{z}|Q')$$

On a alors :

$$\begin{aligned} \hat{P}|\hat{Q} &\equiv \bar{x}.(\nu z)[z](z.R|P')|x.(\lambda z)(\bar{z}|Q') \\ &\rightarrow (\nu z)((z.R|P')|(\bar{z}|Q')) \\ &\rightarrow P'|R|Q' \end{aligned}$$

On aimerait donc avoir pour tout processus d'ordre supérieur  $P$  et  $Q$ ,

$$P \sim Q \iff \hat{P} \sim \hat{Q}$$

pour une congruence naturelle  $\sim$ .

Pour de subtiles raisons (cf [6]), cette propriété n'est pas vérifiée pour certaines congruences et Milner introduit donc l'ordre supérieur directement dans la syntaxe :

### Notation 5.5.1 (Variables d'abstractions)

On note  $X, Y, \dots$  les variables d'abstractions.

La nouvelle syntaxe devient donc :

### Règles syntaxiques 5.5.1

$$\begin{aligned} \text{Processus Normaux : } N &::= \alpha.A \mid \mathbf{0} \mid M + N \\ \text{Processus : } P &::= N \mid P|Q \mid !P \mid (\nu x)P \mid F \\ \text{Abstractions : } F &::= P \mid (\lambda x)F \mid (\lambda X)F \mid (\nu x)F \mid X \mid Fx \mid FG \\ \text{Concrétions : } C &::= P \mid [x]C \mid [F]C \mid (\nu x)C \\ \text{Agents : } A &::= F \mid C \end{aligned}$$

Milner montre que cette extension n'engendre pas de problème particulier (si ce n'est quelques règles ou définitions supplémentaires). Nous ne rentrerons pas dans les détails du  $\pi$ -calcul d'ordre supérieur et terminons avec les sortes d'ordre supérieur.

Milner introduit la notion de *sorte de données* :

### Définition 5.5.1 (Sorte de données)

$$\begin{aligned} \text{Dat}(\mathcal{S}) &\stackrel{\text{def}}{=} \mathcal{S} \cup \text{Ob}(\mathcal{S}) \\ \text{Ob}(\mathcal{S}) &\stackrel{\text{def}}{=} \text{Dat}(\mathcal{S})^* \end{aligned}$$

Ainsi, une sorte d'objet peut être :  $(S_1(S_2S_1)S_3)$ .

**Définition 5.5.2 (Règles de typage)**

On rajoute à la définition 5.4.4 les règles suivantes :

$$\frac{X : S \quad F : t}{(\lambda X)F : (s)\hat{t}}$$
$$\frac{F : (S)\hat{t} \quad x : S}{Fx : t} \quad \frac{F : (s)\hat{t} \quad G : s}{FG : t}$$
$$\frac{F : s \quad C : t}{[F]C : (s)\hat{t}}$$

Notre introduction s'achève donc et nous allons pouvoir utiliser le  $\pi$ -calcul pour donner notre modèle formel.

# Chapitre 6

## Modélisation de notre système d'agents

Ce chapitre est consacré à notre modèle. Après en avoir donné le formalisme, nous montrerons et expliquerons quelques unes de ses limites. Enfin, le modèle sera mis en oeuvre sur un exemple.

### 6.1 Modélisation

Avant d'entrer dans les détails du formalisme, rappelons brièvement les caractéristiques de notre modélisation :

#### Définition 6.1.1 (Caractéristique 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 identifiant local valide pour faire référence à un agent du réseau.

En prenant en compte les règles 1 et 2, un réseau  $\mathcal{R} = S_1, \dots, S_n$  où  $S_i$  est un système d'agents contenant  $k_i$  agents  $A_{i,1}, \dots, A_{i,k_i}$  s'écrit en  $\pi$ -calcul :

$$\begin{array}{l} S_1 \quad | \quad \{A_{1,1}|A_{1,2}|\dots|A_{1,k_1}\} | \\ S_2 \quad | \quad \{A_{2,1}|A_{2,2}|\dots|A_{2,k_2}\} | \\ \vdots \quad | \quad \vdots \\ S_n \quad | \quad \{A_{n,1}|A_{n,2}|\dots|A_{n,k_n}\} \end{array}$$

En programmation orienté objet, le terme de fonction est abandonné au profit de la notion de *méthode* que nous utiliserons aussi. Avant d'aller plus loin, une autre notation nous sera utile :

#### Notation 6.1.1 (Factorisation des noms de méthodes)

Soit  $\{\tilde{X}\#x_1, \dots, \tilde{X}\#x_n\}$  un ensemble de nom de méthode. On notera  $\tilde{X}$  cet ensemble<sup>1</sup>. On écrira  $\tilde{X} \asymp \{x_1, \dots, x_n\}$  pour décrire  $\tilde{X}$ . Par souci de clarté, nous écrirons  $x_i$  pour faire référence à la méthode  $\tilde{X}\#x_i$  lorsque  $\tilde{X}$  sera évident. Enfin,

$$\overline{\tilde{X}\#x_i} = \tilde{X}\#\overline{x_i}$$

Observons sur un exemple comment le  $\pi$ -calcul permet une modélisation intuitive de code Java.

#### Exemple 6.1.1 (Modélisation de code Java en $\pi$ -calcul)

Considérons le fragment de code 6.1.1.

Cette classe peut-être formaliser ainsi :

---

1. On peut considérer le symbole  $\#$  comme l'opérateur de concaténation.

---

**Fragment de code java 6.1.1** Une classe Java

---

```
public class Classe{
    public int ma(int xa){
        return plus(x,2);
    }
    public type mb(type xb){
        ...
    }
    .
    .
    .
    public type mz(type xz){
        ...
    }
}
```

---

$$\begin{aligned} \tilde{m} &\asymp \{ma, mb, \dots, mz\} \\ Classe &\equiv (\lambda \tilde{m}) \left\{ \begin{array}{l} \tilde{m}\#ma(\lambda xa, resultat). \\ \quad \overline{plus}[xa, 2, resultat] \\ + \\ \tilde{m}\#mb(\lambda xb, resultat). \\ \quad \dots \\ + \\ \vdots \\ + \\ \tilde{m}\#mz(\lambda xz, resultat). \\ \quad \dots \end{array} \right\} \end{aligned}$$

Le code:

```
Classe o = new Classe();
int y = plus(o.ma(3), 2);
```

est formalisé par l'expression :

$$(\nu \tilde{m}) \left\{ (Classe)(\tilde{m}) \mid (\nu y, res) \left[ \tilde{m}\#\overline{ma}[3, res] \mid \overline{plus}[res, 2, y] \right] \right\}$$

L'accès à une variable ou une méthode est modéliser 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 :

$$y(\lambda x)$$

.

Pour commencer, nous allons formaliser le serveur. Un serveur est un ensemble de services accessibles depuis l'extérieur et un ensemble d'agents accessibles seulement par leur *id* respectif. Les services sont :

*createAgent, sendAgent, receiveAgent, KillAgent.*

Deux noms nous seront nécessaires pour la gestion de l'ensemble des agents:

*NewAgent, DeleteAgent.*

Ces deux noms seront internes au serveur et ne seront pas accessibles par d'autres objets du réseau. On peut considérer qu'ils représentent les services de la table de correspondance de notre implémentation.

Pour un agent, nous allons séparer le code de ses méthodes incontrôlables - qu'il est le seul à connaître a priori - de leur noms, connus aussi du serveur. 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"<sup>3</sup>, son  $id$  et le serveur créateur. De plus, elle doit envoyer un signal de terminaison sur un canal spécial. Donc ( $|C| = 4$ ). De même,  $R$  ( $|R| = 0$ ) est l'action de l'agent,  $M$  ( $|M_o| = 2$ ) est associé à une action effectuée lorsque l'agent a migré, le nouveau serveur hôte, et le signal de terminaison étant passés en argument,  $M_i$  ( $|M_i| = 1$ ) est associé à l'action à effectuer juste avant de migrer, le serveur destination lui est passé en argument, et enfin,  $D$  ( $|D| = 0$ ) est associé à l'action à effectuer juste avant la mort de l'agent.

$Extra$  est un peu particulière. Cette abstraction d'arité quelconque (cf 6.2.4) 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  $\pi$ -calcul dans  $Extra$ . Dans la suite, nous ne considérerons plus  $Extra$  sauf mention particulière<sup>4</sup>.

Nous l'avons vu, les abstractions représentent le code des méthodes de l'agent. Il leur est associé un nom (sauf pour  $R$  - nous comprendrons pourquoi un peu plus loin). La correspondance entre nom et abstraction est donnée dans le tableau 6.1.

Abstraction	Nom
$C$	<i>onCreation</i>
$M_i$	<i>onMigrating</i>
$M_o$	<i>onMigration</i>
$D$	<i>onDisposing</i>

TAB. 6.1: Correspondance entre noms et abstractions

Les noms de méthode non-associés à une abstraction (*migrate* par exemple) sont des méthodes 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 6.1.2

Un serveur est un ensemble de noms  $\tilde{S} \simeq \{createAgent, sendAgent, receiveAgent, killAgent\}$ . Un agent est composé de deux parties :

- $\widetilde{AgentCode} \simeq \{C, R, M_o, M_i, D, Extra\}$
- $\widetilde{AgentMethod} \simeq \{onCreation, onMigration, onMigrating, onDisposing, migrate, dispose\}$

2. La méthode *exit* de notre implémentation présente peu d'intérêt à être modéliser.

3. Cet argument "mystérieux" sera démasqué en 6.2.4.

4. Nous verrons en 6.2.1 que  $Extra$  représente entre autres, l'état de l'agent.

### 6.1.1 Modélisation du serveur

On définit la création d'un serveur par :

$$\begin{aligned} Server \equiv & (\lambda \tilde{S}) \\ & (\nu NewAgent, DeleteAgent) \\ & !((Agents)(\tilde{S}, NewAgent, DeleteAgent) \\ & |(Services)(\tilde{S}, NewAgent, DeleteAgent)) \end{aligned}$$

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 :

$$(\nu \tilde{S})(Server)(\tilde{S})$$

*Agents* est défini comme suit :

$$\begin{aligned} Agents \equiv & (\lambda \tilde{S}, NewAgent, DeleteAgent). \left\{ \right. \\ & NewAgent(\lambda id, \widetilde{AgentCode}). \\ & (\nu \widetilde{AgentMethod}) \\ & \left. \left\{ (Agent)(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \tilde{S}) | \overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}] \right\} \right. \\ & + \\ & DeleteAgent(\lambda id). \\ & id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\ & \left. \left. \widetilde{AgentMethod} \# \overline{onDisposing} \right\} \right. \end{aligned}$$

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 *AgentMethod* et par l'application de l'abstraction *Agent* (d'arité quatre) et du quadruplet  $(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \tilde{S})$ . Ainsi, l'agent connaît :

- l'ensemble *AgentMethod* de ses noms de méthodes;
- son identificateur *id* sur ce serveur;
- les méthodes  $\tilde{S}$  du serveur sur lequel il réside : c'est son serveur local.

La création s'achève avec la mise à disposition à tous objets les connaissant *id* du code et du nom des méthodes de l'agent. 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 (6.2.2).

La destruction d'un agent est triviale. 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.

Les services représentent l'ensemble des noms de méthodes publiques (dans le sens où elles sont accessibles à quiconque). L'une de ces méthodes est *createAgent*. Elle reçoit le code de l'agent, 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 :

$$\begin{aligned} & \tilde{S} \# createAgent(\lambda \widetilde{AgentCode}, arg, getid). \\ & (\nu id) \overline{NewAgent}[id, \widetilde{AgentCode}]. \\ & id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\ & \overline{AgentMethod} \# \overline{onCreation}[arg]. \\ & \overline{getid}[id] \end{aligned}$$

De même, l'envoi d'un agent par un serveur  $\tilde{S}$  sur un serveur  $\tilde{S}'$  passe par plusieurs étapes :

- recevoir l'identificateur de l'agent à envoyer via le nom de méthode  $\tilde{S}\#sendAgent$  ;
- appeler la méthode *onMigrating* de l'agent en question;
- appeler la méthode  $\tilde{S}'\#receiveAgent$  du serveur récepteur;
- renvoyer l'*id* de l'agent dans son nouveau serveur local.

Par conséquent, nous aurons :

$$\begin{aligned} & \tilde{S}\#sendAgent(\lambda id, \tilde{S}', getNewid). \\ & \quad id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\ & \quad \widetilde{AgentMethod}\#onMigrating[\tilde{S}']. \\ & \quad (\nu getid)\tilde{S}'\#receiveAgent[\widetilde{AgentCode}, getid]. \\ & \quad \quad \frac{getid(\lambda Newid).}{getNewid[Newid]} \end{aligned}$$

Les deux autres méthodes *receiveAgent* et *killAgent* sont similaires. Voici la modélisation complète du serveur :

$$\begin{aligned}
Server &\equiv (\lambda \tilde{S}). \left\{ \begin{array}{l}
(\nu NewAgent, DeleteAgent) \\
!((Agents)(\tilde{S}, NewAgent, DeleteAgent) \\
|(Services)(\tilde{S}, NewAgent, DeleteAgent))
\end{array} \right\} \\
Agents &\equiv (\lambda \tilde{S}, NewAgent, DeleteAgent). \left\{ \begin{array}{l}
NewAgent(\lambda id, \widetilde{AgentCode}). \\
(\nu AgentMethod) \\
\left\{ (Agent)(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \tilde{S}) | \overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}] \right\} \\
+ \\
DeleteAgent(\lambda id). \\
id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
AgentMethod\#onDisposing
\end{array} \right\} \\
Services &\equiv (\lambda \tilde{S}). \left\{ \begin{array}{l}
\tilde{S}\#createAgent(\lambda \widetilde{AgentCode}, arg, getid). \\
(\nu id)\overline{NewAgent}[id, \widetilde{AgentCode}]. \\
id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
\overline{AgentMethod\#onCreation}[arg]. \\
getid[id] \\
+ \\
\tilde{S}\#sendAgent(\lambda id, \tilde{S}', getNewid). \\
id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
AgentMethod\#onMigrating[\tilde{S}']. \\
(\nu getid)\tilde{S}'\#\overline{receiveAgent}[\widetilde{AgentCode}, getid]. \\
getid(\lambda Newid). \\
getNewid[Newid] \\
+ \\
\tilde{S}\#receiveAgent(\lambda \widetilde{AgentCode}, getid). \\
(\nu id)\tilde{S}\#\overline{NewAgent}[id, \widetilde{AgentCode}]. \\
id(\lambda \widetilde{AgentCode}, \widetilde{AgentMethod}). \\
\overline{AgentMethod\#onMigration}. \\
getid[id] \\
+ \\
\tilde{S}\#killAgent(\lambda id). \\
\tilde{S}\#\overline{DeleteAgent}[id]
\end{array} \right\}
\end{aligned}$$

## 6.1.2 Modélisation de l'agent

Comme nous l'avons vu, un agent doit pouvoir communiquer avec le serveur, pour migrer par exemple. Il doit aussi pouvoir communiquer avec les objets extérieurs via son *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 *à priori*<sup>5</sup> contrôler le nombre de leurs appels. Nous donnons maintenant le formalisme de l'agent puis nous le commentons.

$$\begin{aligned}
 \text{Agent} \equiv & (\lambda \widetilde{\text{AgentCode}}, \widetilde{\text{AgentMethod}}, id, \widetilde{S}) \left\{ \right. \\
 & \left\{ (\nu \text{run}) [ \right. \\
 & \quad \widetilde{\text{AgentMethod}}\#onCreation(\lambda arg).(\widetilde{\text{AgentCode}}\#C)(arg, id, \widetilde{S}, \text{run}) \\
 & \quad + \\
 & \quad \widetilde{\text{AgentMethod}}\#onMigration.(\widetilde{\text{AgentCode}}\#M_o)(\widetilde{S}, \text{run}) \\
 & \quad \left. \right] | \text{run}.\widetilde{\text{AgentCode}}\#R \left\} \right. \\
 & | [\widetilde{\text{AgentMethod}}\#onMigrating(\lambda \widetilde{S}').(\widetilde{\text{AgentCode}}\#M_i)(\widetilde{S}') \\
 & \quad + \\
 & \quad \widetilde{\text{AgentMethod}}\#onDisposing.(\widetilde{\text{AgentCode}}\#D)] \\
 & | [\widetilde{\text{AgentMethod}}\#migrate(\lambda \widetilde{S}').\widetilde{S}\#\overline{\text{sendAgent}}[id, \widetilde{S}'] \\
 & \quad + \\
 & \quad \widetilde{\text{AgentMethod}}\#dispose.\widetilde{S}\#\overline{\text{killAgent}}[id] \\
 & \left. \right\}
 \end{aligned}$$

La méthode *onCreation* reçoit un argument "mystérieux" (cf 6.2.4), applique l'abstraction *C* puis lance le processus *R* en parallèle lorsqu'un signal de terminaison à été reçu sur le nom *run* assurant la séquentialité<sup>6</sup>. *R* doit être appliqué en parallèle car d'une part, cela correspond à l'allocation d'un thread dans notre implémentation, et d'autre part, *C* peut mettre à jour des variables que *R* souhaite utiliser. nous avons vu dans la remarque de l'exemple 6.1.1 que 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*.

## 6.2 Étude de notre modèle

Nous allons étudier les propriétés de notre modèle et déterminer les problèmes qu'il pose.

### 6.2.1 L'état de l'agent

Souvenons-nous qu'un agent est avant tout un processus et que c'est donc un ensemble de threads et de données. A un instant donnée, 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<sub>o</sub>*, *M<sub>i</sub>*, *D* doivent pouvoir communiquer avec *Extra* (Condition sine qua non en  $\pi$ -calcul). *C* devra donc exécuter *Extra* en parallèle pour permettre aux autres abstractions de l'utiliser.

5. Nous verrons que ce n'est pas le cas en 6.2.2

6. Nous verrons que ce n'est pas le cas dans la section 6.2.2

## 6.2.2 Problème du 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  $R$ . Même les abstractions  $C$ ,  $M_o$ ,  $M_i$  et  $D$  peuvent faire des choses totalement inattendues. Par exemple, on peut avoir  $C \equiv \overline{run}.C'$ . Dans ce cas, un signal de terminaison va être reçu, le processus  $R$  lancé alors qu'il reste  $C'$  à exécuter. Ce problème est tout à fait conforme à la réalité. Il n'est pas possible de garantir d'un processus que l'on ne connaît pas qu'il effectuera une action déterminée avant une autre. Ce point a été discuté dans notre implémentation section 3.2.1.

De plus, 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 son serveur local 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. En effet, il suffit de différencier les méthodes destinée au serveur des autres par des  $id$  différents. On pourrait avoir par exemple  $id_{M_s}$  pour l'accès au méthodes dédiées au serveur,  $id_{M_a}$  pour celle dédiée 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. Toutefois, cette démarche exige une implémentation un peu plus lourde que celle qui est proposée et c'est dans un soucis de cohérence avec l'implémentation que cette solution n'a pas été retenue. Cependant, nous avons montré comment résoudre ce problème et nous montrerons de même en 3.2.1 comment implémenter cette solution.

## 6.2.3 Problème de destruction d'un agent

Le  $\pi$ -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<sup>7</sup>. Pire, l'abstraction  $R$  par exemple, inconnue du serveur peut contenir une ou plusieurs réplifications. Il peut donc y avoir accumulation de termes dans l'expression au fur et à mesure des créations d'agents.

Toujours est-il que ce problème est conforme à l'implémentation. Ce problème à déjà été discuté dans la section 3.2.2.

## 6.2.4 Problème polymorphe

Le nom de méthode  $onCreation$  reçoit un argument “utilisateur” et le passe à  $C$  :

$$\dots (\widetilde{AgentMethod}\#onCreation(\lambda arg).(\widetilde{AgentCode}\#C)(arg, id, \tilde{S}, run)) \dots$$

Cet argument est “mystérieux” 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.

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).

De plus, il n'est pas possible de définir une fois pour toutes de qu'elle sorte sera le nom *onCreation* ou l'abstraction *C*. En effet, un agent peut avoir besoin d'un vecteur de taille  $n$  et d'un autre de taille  $n + 1$ . La même remarque s'applique à l'abstraction *Extra*.

La solution est d'étendre le  $\pi$ -calcul est d'y introduire le polymorphisme. Nous ne rentrerons pas dans les détails de cette extension mais nous utiliserons juste les résultats de [25]. Si dans le formalisme donné, nous n'avons pas introduit la notation polymorphique, c'est pour d'évidentes raisons de clarté.

## Chapitre 7

# Proposition d'une généralisation : les conteneurs actifs

Nous avons vu comment modéliser un système d'agents et comment l'implémenter en Java. Cependant, il est possible d'aller plus loin dans notre étude mais en sortant un peu du cadre des agents pour introduire une notion plus générale. En premier lieu, nous proposons une nouvelle façon de se représenter un système d'agents. Cette nouvelle représentation nous est venue à l'esprit en nous interrogeant sur la raison pour laquelle la création d'un agent ne peut-être faite que par un serveur. Se poser cette question revient à se demander ce qu'est réellement un serveur d'agents. En réalité, au niveau des systèmes d'agents, la notion de migration d'un agent peut-être remplacée par deux opérations de niveau inférieur :

- retirer l'agent de la vue de son serveur local,
- insérer l'agent dans la vue du serveur destinataire de la migration.

Nous pouvons donc utiliser la notion de conteneur pour simuler celle de serveur d'agents. Un objet étant associé à un *id* comme dans notre système d'agents.

Par ailleurs, dans notre serveur d'agents, chaque méthodes susceptibles de modifier l'état de sa table de correspondance (`createAgent`, `exit`, `sendAgent`, `receiveAgent` et `callAgentMethod`) appelle une méthode d'un agent (`onCreation`, `onDisposing`, `onMigrating`, `onMigration`). Cette caractéristique peut-être donné à un conteneur :

### **Définition 7.0.1 (Caractéristique de notre conteneur)**

*Toutes méthodes susceptibles de modifier la table de correspondance d'un conteneur nécessite l'appel d'une méthode sur un ou plusieurs de ses objets.*

## 7.1 Le conteneur actif

Notre nouvel objet est donc un peu plus qu'un simple conteneur puisqu'il stocke des objets et appellent leur méthodes. Remarquons qu'il n'y a plus l'idée de migration, ni même d'agent.

### **Définition 7.1.1 (Conteneur actif)**

*Un conteneur actif est un conteneur dont une modification de son contenu implique l'allocation d'un thread pour l'appel d'une méthode d'un de ses objets.*

## 7.2 Implémentation

L'implémentation est on ne peut plus simple. L'interface du serveur est donnée en 7.2.1. On peut remarquer que notre conteneur implémente RMI devenant ainsi un conteneur actif distant.

La caractéristique vu en 7.0.1 est clairement assuré ici : chaque méthode susceptibles de modifier la table de correspondance de notre conteneur actif prend en argument une méthode `method` nécessairement serialisable puisque

---

**Fragment de code java 7.2.1 L'interface du conteneur actif**

---

```
package agents.model2;

import java.io.Serializable;

public interface ActiveStockerServer extends java.rmi.Remote {
    public MethodResult put(Serializable objectPart,
        SerializableMethod method,
        Serializable args[],
        boolean waitResult)
        throws java.rmi.RemoteException;

    public Serializable get(String id) throws java.rmi.RemoteException;

    public MethodResult remove(String id,
        SerializableMethod method,
        Serializable args[],
        boolean waitResult)
        throws java.rmi.RemoteException;

    public MethodResult callMethod(String id,
        SerializableMethod method,
        Serializable args[],
        boolean waitResult)
        throws java.rmi.RemoteException;

    public java.util.Hashtable getAll() throws java.rmi.RemoteException;

    public java.util.Enumeration getAllId() throws java.rmi.RemoteException;

    public String getHost() throws java.rmi.RemoteException;

    public String getServerAddress() throws java.rmi.RemoteException;

    public void exit() throws java.rmi.RemoteException;
}
```

---

nous utilisons `RMI` qui copie ses arguments (cf. 3.2.3)<sup>1</sup>. Cette méthode sera exécutée dans une nouvelle thread. L'argument `waitResult` permet à l'appelant de spécifier s'il doit être bloqué et attendre la fin de la thread allouée (retour de `method` ou exception) ou si au contraire, il ne faut pas s'en soucier et retourner immédiatement. Enfin, la classe `MethodResult` contient l'objet (valeur de retour ou exception) retourné par `method` et `id` associé à l'objet visé par l'appel de `method`.

Les problèmes liés à la destruction des objets et des threads en Java se répercutent automatiquement sur cet objet serveur. Remarquons qu'il n'y a pas de méthode équivalente à `create`. Cette notion n'a plus de sens puisqu'un objet est simplement inséré ou retiré du conteneur.

### 7.3 Modélisation

Nous reprenons le même schéma que pour le serveur d'agents. Notre conteneur nécessite deux noms de méthodes : `put` et `remove` respectivement pour insérer et retirer un objet. De la même manière que pour les agents, un objet possède un `id`. C'est sa référence pour le conteneur. Cet `id` a les mêmes propriétés que celui des agents.

On définit donc un conteneur actif par :

$$\widetilde{CS} \simeq \{put, remove\}$$

On a donc formellement en  $\pi$ -calcul :

$$\begin{aligned} ActiveStocker \equiv & (\lambda \widetilde{AS}). \\ & (\nu New, Delete) \\ & !((Objects)(\widetilde{AS}, New, Delete) \\ & | (Services)(\widetilde{AS}, New, Delete)) \end{aligned}$$

Le conteneur proprement dit *Objects* est défini ainsi :

$$\begin{aligned} Objects \equiv & (\lambda \widetilde{AS}, New, Delete). \left\{ \right. \\ & New(\lambda id, F, \widetilde{O}). \\ & \quad (F)(id, \widetilde{O}) | !\overline{id}[\widetilde{O}] \\ & + \\ & Delete(\lambda id). \\ & \quad \underbrace{\{!id(\lambda \widetilde{O})\}}_{\text{Inutile !}} \\ & \left. \right\} \end{aligned}$$

L'expression annotée "Inutile !" illustre la difficulté qu'il y a en Java de détruire un objet ou un Thread. Intuitivement, nous pouvons penser que pour éliminer l'emploi d'un terme répliqué ( $!\overline{id}[\widetilde{C}]$  dans notre cas), il suffit de répliquer son sujet complémentaire comme dans le processus :

$$P \equiv !x(\lambda y) | (\nu z) \overline{x}[z]$$

afin de consommer les messages émis. Malheureusement, si  $Q \equiv x(\lambda w)$ , alors dans l'expression  $P|Q$ ,  $Q$  recevra bien sur le canal de nom  $x$  un nom  $z$ . Souvenons-nous en effet que  $!P \stackrel{\text{def}}{=} P|!P$ . Malgré tout, nous garderons cette expression en l'état au cas où le problème de destruction d'objet ou de Thread serait résolu. Dans ce cas, il suffirait de modifier légèrement le modèle pour l'adapter à l'implémentation.

1. La classe `SerializableMethod` contourne le fait que `java.lang.reflect.Method` n'est pas serialisable.

Si  $F$  et  $\tilde{O}$  représentent l'objet à stocker<sup>2</sup> les services sont définis ainsi :

$$\begin{aligned}
\text{Services} \equiv & (\lambda \tilde{A}S, \text{New}, \text{Delete}). \left\{ \right. \\
& \tilde{A}S \# \overline{\text{put}}(\lambda F, \tilde{O}, \text{method}, \text{arg}, \text{waitResult}, \text{result}). \\
& (\nu \text{id}) \overline{\text{New}}[\text{id}, F, \tilde{O}]. \\
& \quad (\nu \text{res}) \overline{\text{method}}[\text{arg}, \text{res}]. \\
& \quad (\nu t, f) (\text{waitResult})(t, f) \\
& \quad \quad | t.\text{res}(\lambda \text{val}).\overline{\text{result}}[\text{id}, \text{val}] \\
& \quad \quad | f.\overline{\text{result}}[\text{id}] \\
& + \\
& \tilde{A}S \# \overline{\text{remove}}(\lambda \text{id}, \text{method}, \text{arg}, \text{waitResult}, \text{result}). \\
& \overline{\text{Delete}}[\text{id}]. \\
& (\nu \text{res}) \overline{\text{method}}[\text{arg}, \text{res}]. \\
& \quad (\nu t, f) (\text{waitResult})(t, f) \\
& \quad \quad | t.\text{res}(\lambda \text{val}).\overline{\text{result}}[\text{id}, \text{val}] \\
& \quad \quad | f.\overline{\text{result}}[\text{id}] \\
& \left. \right\}
\end{aligned}$$

$\text{method}$  est une méthode de  $\tilde{O}$  qui prend un argument  $\text{arg}$ .  $\text{waitResult}$  est une abstraction qui représente une variable booléenne. Si elle est à vrai, alors on attend la valeur de retour de la méthode invoquée que l'on retourne en plus de l' $\text{id}$  alloué. Sinon, on retourne directement l' $\text{id}$  alloué.

## 7.4 Simulation d'un système d'agent

Nous pouvons utiliser notre conteneur actif pour simuler notre système d'agent. En effet, on peut considérer que notre conteneur actif est la couche bas niveau d'un serveur d'agent. La couche de plus haut niveau offrira les services que nous avons modélisé à savoir :

$$\tilde{S} \asymp \{\text{create}, \text{send}, \text{receive}, \text{kill}\}$$

Considérons l'ensemble :

$$\tilde{A} \asymp \{\text{init}, \text{onCreation}, C, \text{onMigration}, M_o, \text{onMigrating}, M_i, \text{onDisposing}, D, R\}$$

Notre serveur d'agent s'écrit donc :

---

2.  $F$  représente le code et  $\tilde{O}$  les méthodes de l'objet à stocker



Il serait intéressant de prouver que l'on peut simuler notre système d'agents par ce modèle que nous qualifierons de "bas niveau". De même, est-il possible de simuler le système des aglets d'IBM ou le système Odyssey de General Magic. Pour répondre à ces questions, des outils sont nécessaires pour trouver les relations de simulations adéquates - si elles existent. De tels outils n'ont pas été trouvés. Certains sont en cours de développement notamment dans l'équipe de Tom Melham [42]. Malgré tout, intuitivement, nous pensons que ces simulations sont possibles avec notre système de conteneur actif.

# Chapitre 8

## Conclusion

Nous avons vu une implémentation en Java d'un système d'agents et sa modélisation en  $\pi$ -calcul. Nous avons aussi vu les limites de notre implémentation - limites que souligne notre modélisation. Certaines d'entre elles sont liées au langage (destruction de threads par exemple), d'autres au protocole utilisé i.e RMI (copies des paramètres et des valeurs de retour des méthodes). Par ailleurs, nous avons montré qu'un nouvel objet nommé *conteneur actif* permettait de simuler notre système d'agents. Nous pensons que la plupart des systèmes d'agents existants (vu au chapitre 2) sont simulables par notre conteneur actif mais cela reste à démontrer. Une telle preuve nécessite l'emploi d'outils qui permettent de raisonner sur les expressions en  $\pi$ -calcul. Ces outils sont en cours de développement comme par exemple les travaux de Tom Melham [41] déjà vu en 4.1.5.

Toutefois, l'utilisation d'un autre formalisme comme les ambiances mobiles ([7]) permet peut-être une modélisation et une démonstration plus aisées.

Enfin, nous nous interrogeons sur les possibilités de notre conteneur actif. Plusieurs idées viennent à l'esprit comme le stockage d'objets à distance ou le parallélisme massif en utilisant les machines connectées sur Internet. Il pourrait être intéressant de s'intéresser à ce type d'objets pour comprendre ce qu'il peuvent apporter aux applications distribuées en général et au projet JEM en particulier.

# Bibliographie

- [1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–??, 1997.
- [2] The Agent Society. *Design Workshop on Open Intelligent Agent Platforms and Protocols*, 9-10 février 1997. First Meeting of the Agent Interop Working Group.
- [3] Ken Arnold and James Gosling. *The Java programming language*. Addison-Wesley, 1996.
- [4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.
- [5] Gérard Boudol. Asynchrony and the  $\pi$ -calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.
- [6] B.Thomsen. *Calculi for higher-order communicating systems*. PhD thesis, Imperial College, London University, 1990.
- [7] 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.
- [8] Serge Chaumette. Experimentation environMent for Java – Un environnement Java distribué pour le développement et l'expérimentation. Submitted to NOTERE'98, 1998.
- [9] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951.
- [10] Jim White Cynthia Tham, Barry Friedman. Maf issues. Technical report, General Magic Inc., 25 novembre 1996.
- [11] Jim White Cynthia Tham, Barry Friedman and Tony Rutkowski. An assessment of the mobile agent facility proposal. Technical report, General Magic Inc., 10 janvier 1997.
- [12] M.Abadi et A.Gordon. A calculus for cryptographic protocols - the spi calculus. Page Web, 11 décembre 1996.
- [13] 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.
- [14] the Open Group GMD FOKUS, IBM Corporation. *Mobile Agent Facility Specification*, 15 décembre 1996.
- [15] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. *Lecture Notes in Computer Science*, 512:133–??, 1991.
- [16] F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, December 1995. (Also published at Technical Report CMU–CS–95–223.).
- [17] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Principles of Programming Languages*, 1996.

- [18] Danny B. Lange and Yariv Aridor. *Agent Transfer Protocol – ATP/0.1*. IBM Corporation, 19 mars 1997.
- [19] Danny B. Lange and Mitsuru Oshima. *Programming Mobile Agent in Java - With the Java Aglet API*. IBM Research, 1997.
- [20] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
- [21] R. Milner. *Lectures on a Calculus for Communicating Systems*, volume 197 of LNCS. Springer-Verlag, New York, NY, 1984.
- [22] Robin Milner. The polyadic  $\pi$ -calculus: a tutorial. Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, octobre 1991.
- [23] Mitsuru Oshima and Guenter Karjoth. *Aglets Specification (Alpha5) Draft*. IBM Corporation, 10 septembre 1997.
- [24] 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.
- [25] 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.
- [26] Aglets (IBM Corporation). <http://www.tr1.ibm.co.jp/aglets/>.
- [27] Concordia (Mitsubishi Electric). <http://www.meitca.com/hsl/projects/concordia/>.
- [28] GeneralMagic. <http://www.genmagic.com/>.
- [29] International Conferences, Exhibitions on the Practical Application of Intelligent Agents, and Multi-Agents. <http://www.demon.co.uk/ar/>.
- [30] Liens sur le calculs de processus mobiles (Uwe Nestmann). <http://www.cs.auc.dk/mobility/>.
- [31] Liste de systèmes d'agents (W3C). <http://www.w3.org/mobilecode/>.
- [32] Object Management Group. <http://www.omg.org/>.
- [33] Odyssey (General Magic). [http://www.genmagic.com/technology/mobile\\_agent.html](http://www.genmagic.com/technology/mobile_agent.html).
- [34] Selected Bibliography on Mobile Processes (Kohei Honda). <http://www.cs.auc.dk/mobility/bib/honda.html>.
- [35] the Agent Society. <http://www.agents.org/>.
- [36] Voyager (ObjectSpace). <http://www.objectspace.com/voyager/>.
- [37] Wave (Sapaty). <http://www.cs.dartmouth.edu/~agent/workshop/1996/abstracts/sapaty>.
- [38] 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.
- [39] Davide Sangiorgi.  $\pi$ -calculus, internal mobility and agent-passing calculi. Technical Report RR-2539, INRIA-Sophia Antipolis, 1995. To appear in *Theoretical Computer Science*; an abstract appeared in TAPSOFT'95.
- [40] Jon Siegel. *CORBA, Fundamental and Programming*. Wiley, 1996.
- [41] T.F. Melham. *Introduction to the HOL theorem prover*. University of Cambridge, Computer Laboratory, Cambridge, England, 1990.
- [42] 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.
- [43] Bent Thomsen. *Calculi for Higher Order Communicating Systems*. Ph.D. thesis, Imperial College, London, 1990.

- [44] 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.
- [45] U.Engberg and M.Nielsen. A calculus of communicating system with label-passing. Technical report, Computer Science Department, University of Aarhus, Denmark, 1996.
- [46] Jim White. Ibm tokyo labs-general magic meeting. Sunnyvale CA, 1 novembre 1996.