Titre: Title:	Conception d'une architecture multi-agents supportant des agents mobiles intelligents
Auteur: Author:	Sylvain Goutet
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Goutet, S. (2001). Conception d'une architecture multi-agents supportant des agents mobiles intelligents [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6960/

UNIVERSITÉ DE MONTRÉAL

CONCEPTION D'UNE ARCHITECTURE MULTI-AGENTS SUPPORTANT DES AGENTS MOBILES INTELLIGENTS

SYLVAIN GOUTET
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AVRIL 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

CONCEPTION D'UNE ARCHITECTURE MULTI-AGENTS SUPPORTANT DES AGENTS MOBILES INTELLIGENTS

présenté par : GOUTET Sylvain

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen composé de:

M. ROY Robert, Ph. D., Président

M. PIERRE Samuel, Ph. D., membre et directeur de recherche

M. GLITHO Roch, M. Sc., membre et codirecteur de recherche

M. QUINTERO Alejandro, Ph. D., membre

Remerciements

Je désire remercier mon directeur de recherche, le professeur Samuel Pierre, et mon co-directeur, Roch Glitho, pour leur patience, leurs conseils et leurs commentaires.

Je désire ensuite remercier tous les membres du Laboratoire de Recherche en Réseautique et Informatique Mobile (LARIM), pour leur aide et leurs critiques.

Je désire également remercier ma famille et mes amis pour leurs encouragements et leur soutien.

Résumé

Depuis quelques années, on assiste – et participe - à l'accroissement spectaculaire des réseaux, et d'Internet en particulier, tant en quantité de données présentées et échangées, qu'en nombre d'utilisateurs ou en étendue géographique. Cette expansion s'est appuyée sur la technologie client/serveur qui a, jusqu'à présent, réussi à s'adapter et à transporter et traiter un volume de données toujours plus grand. Cependant, ce système a aussi montré des faiblesses : sa grande « centralisation » conduit à un problème d'évolutivité, de manque de personnalisation et d'un manque de prise en compte de la topologie du monde réel et «virtuel». Ce dernier problème n'est toutefois pas imputable uniquement à la technologie client/serveur, mais aux choix faits à la création d'Internet. C'est pourquoi on fonde beaucoup d'espoir sur les nouvelles technologies, qui sortent à peine des laboratoires, d'agents et d'agents mobiles. Malgré des défauts certains et le manque d'applications où ils marquent une réelle différence avec les technologies existantes, ces systèmes montrent des potentialités intéressantes pour la flexibilité et la réduction de la charge des réseaux, en particulier des grands réseaux.

Une des particularités, mais aussi un des handicaps des agents mobiles est de devoir transporter à chaque trajet tout le code nécessaire à l'accomplissement de leur tâche, alors que les autres applications ne déplacent que les données. Une approche à ce problème est donnée par un système de cache, comme celui implanté dans le système Grasshopper. Malheureusement, ce que le système gagne en performance est perdu en souplesse. En particulier, le remplacement ou la mise à jour d'agents devient problématique.

Dans ce mémoire, nous proposons une architecture multi-agents qui résout ce problème en séparant l'agent mobile des différents services qu'il va utiliser – interfaces avec les différents systèmes, algorithmes de calcul ou de recherche d'information - et qui sont réalisés par des agents «passifs» pouvant être également mobiles. Notre architecture propose des mécanismes pour permettre la recherche de services et la

communication entre agents qui impliquent un agent «Registraire». Le rôle de cet agent est de procurer à un agent mobile arrivant sur le serveur le service dont il a besoin. Si aucun agent local ne procure ce service, il va chercher le plus proche possible pour le copier ou le charger, ce qui nécessite une idée de voisinage géographique dans le réseau. Cette notion de voisinage est étendue à l'ensemble du réseau sous forme de zone. Nous proposons des algorithmes qui permettent à un agent mobile de tirer profit de ces connaissances sur son environnement pour optimiser ses déplacements.

Des mesures de performance ont permis de prouver la validité de la conception de l'architecture. Nous nous sommes focalisés sur l'apport effectif de l'utilisation de connaissances sur le réseau et d'algorithmes de routage plus complexes au regard de l'augmentation de la taille de l'agent qu'ils entraînent. Nous avons constaté que, du fait de l'utilisation du cache de Grasshopper et de la machine Java, le déplacement d'un agent utilisant l'un ou l'autre des algorithmes est équivalent. Considérant maintenant le nombre de déplacements de l'agent nécessaires pour trouver le bon correspondant avec chaque algorithme, tous mettent à profit le mécanisme d'apprentissage par rétroaction pour réduire leurs déplacements. Les deux algorithmes utilisant en plus des connaissances sur la topologie du réseau montrent un réel avantage. Cependant, les mesures n'ont pas réussi à mettre en évidence une supériorité de l'algorithme le plus complexe par rapport au moins complexe parmi ces deux algorithmes. Même si la comparaison avec une implémentation client/serveur optimale reste au désavantage des agents mobiles, l'architecture multi-agents proposée représente un moyen simple et efficace de pallier les déficiences d'une implémentation client/serveur inadaptée.

Abstract

For the last few years, networks, particularly Internet have increased incredibly, in the amount of data exchanged, number of users, and geographical extension. This expansion was based on the client/server technology which, until now, managed to adapt, carry and treat this ever increasing amount of data. This system showed weaknesses : it is centralized, poorly scalable and personalized, and does not take into account the topology of the real or virtual world. This last problem does not come from the client/server technology itself but from the choices made when Internet was created. That is the reason why many efforts are put on the new technologies, still in laboratories, of agents and mobile agents. In spite of many weaknesses and the lack of killer applications, these systems show interesting capabilities considering flexibility and reduction of network load, specially in large networks.

Mobile agents are based on and, at the same time, handicapped by the fact that they must carry their whole code they need to accomplish their task, where as other applications only send data on the network. One approach of this problem, implemented in the Grasshopper system, is caching. Unfortunately, the system looses some adaptability because a cached agent cannot be replaced by another version of the same agent. In this paper, we propose a multi-agent architecture which solves this problem by splitting the mobile agent in several services, small agents that will cooperate to accomplish the task. The proposed architecture offers mechanisms of search and communication, implemented in the "Registraire" agent. This agent has to give mobile agents the services they need when arriving on a new machine. If no local agent offers a service, the "Registraire" will load the agent from the nearest possible machine. This implies an idea of neighborhood, which is applied to the whole network as "zones". We also propose routing algorithms that use this knowledge.

Performance measures validated the conception of the architecture. We focused on the utility of more complex algorithms that need to carry more data but can be more efficient. We found that all algorithms could benefit from feedback learning algorithms,

and that the algorithms using data on the topology of the network were more efficient. Nevertheless, the measures we made could not show an advantage of the more complex over a less complex of the latest algorithms. Even if an optimized client/server implementation remains the best in terms of performance, the multi-agent architecture we propose represents an efficient way to cope with a bad or inefficient client/server implementation.

Table des Matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

APD	Appel de Procédure à Distance
CNRC	Conseil National de la Recherche du Canada
CTI	Computer Telephony Integration
HTML	HyperText Markup Language
IA	Intelligence Artificielle
IIT	Institute for Information Technology
IUG	Interface Utilisateur Graphique
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulating Language
LAN	Large Area Network
MASIF	Mobile Agent System Interoperability Facility
MVJ	Machine Virtuelle Java
OMG	Object Management Group
ORB	Object Request Broker
RI	Recherche d'Information
SAM	Système d'Agents Mobiles
SIP	Session Initiation Protocol
SPIN	Seamless Personal Information Networking
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WONDEL	Web ONtology Description Language
XML	eXtended Markup Language

Chapitre I
Introduction

Depuis quelques années, on assiste – et participe - à l'accroissement spectaculaire des réseaux, et d'Internet en particulier, tant en quantité de données présentées et échangées, qu'en nombre d'utilisateurs ou en étendue géographique. Cette expansion s'est appuyée sur la technologie client/serveur qui a, jusqu'à présent, réussi à s'adapter, à transporter et traiter un volume de données toujours plus grand. Cependant, ce système a aussi montré des faiblesses : sa grande « centralisation » conduit à un problème d'évolutivité, de manque de personnalisation et d'un manque de prise en compte de la topologie du monde réel et «virtuel». Ce dernier problème n'est toutefois pas imputable uniquement à la technologie client/serveur, mais aux choix faits à la création d'Internet. C'est pourquoi on fonde beaucoup d'espoir sur des nouvelles technologies, qui sortent à peine des laboratoires, d'agents et d'agents mobiles. Malgré des défauts certains et le manque d'applications où ils marquent une réelle différence avec les technologies existantes, ces systèmes montrent des potentialités intéressantes pour la flexibilité et la réduction de la charge des réseaux, en particulier des grands réseaux.

1.1 Définitions et concepts de base

Un agent est communément défini comme une aide logicielle qui remplace l'utilisateur dans une tâche routinière et pénible, comme organiser l'horaire d'une réunion ou le tri du courrier électronique ou qui cherche et trie des informations correspondant aux intérêts de l'utilisateur (Gray, 1995; Hafner, 1995; Rogers, 1995). Les agents se distinguent particulièrement des autres logiciels par leur autonomie vis-à-vis de l'utilisateur. Alors qu'une application classique requiert de la part d'un utilisateur des paramètres précis pour chaque type de situation qu'elle doit rencontrer, l'agent se doit de les « deviner » ou de les extrapoler du comportement de l'utilisateur tout en étant à

même de faire face à des situations imprévues, dans une certaine mesure. Cela suppose une grande part d'intelligence, et, de fait, le concept d'agent est très lié à celui de l'intelligence artificielle.

Les agents mobiles sont des agents possédant en outre la capacité de se déplacer entre deux ou plusieurs nœuds d'un réseau. Bien que leur nom les rapproche des agents fixes, le concept d'agent mobile prend réellement sa source dans celui de processus et de code mobile. Celui-ci cherchait à rendre mobiles des processus de façon à ce que ceux-ci puissent interrompre leur exécution à tout moment pour se transférer sur une autre machine qui, par exemple, dispose de plus de mémoire que la première. Cette capacité demandait habituellement à être soutenue par de complexes systèmes d'exploitation, ce qui a contribué à son échec.

Le concept de code mobile consiste à définir un code tel que les programmes compilés dans ce code puissent être portés et exécutés sur n'importe quelle machine (et même des machines telles qu'une cuisinière ou une chaîne hi-fi, pourvu qu'elles soient dotées du matériel nécessaire). Le code mobile est généralement soutenu par une machine virtuelle qui va traduire et exécuter localement ses instructions. Il a trouvé une grande popularité avec Java, mais il existe également d'autres langages, tels Telescript, Python ou Scheme.

1.2 Éléments de la problématique

Les agents mobiles, comme on peut le voir à leur nom et à leur définition, se posent comme héritiers de deux technologies : les agents fixes et le code mobile. Les agents fixes doivent beaucoup à l'intelligence artificielle. Une grande part de la recherche actuelle sur les agents se concentre sur les systèmes multi-agents, dans lesquels plusieurs agents coopèrent au moyen de messages et d'un langage commun afin de résoudre une même tâche. Ces recherches visent à obtenir ainsi une forme d'intelligence distribuée. Le code mobile doit son succès à celui des télécommunications, en particulier d'Internet. Ces deux domaines sont habituellement

bien séparés. Les agents mobiles (ou leurs concepteurs) veulent ainsi pouvoir réunir le meilleur des deux concepts, mais se retrouvent souvent avec les défauts des deux : la lourdeur de l'intelligence artificielle et les limitations des communications. Les applications d'agents mobiles présentées par l'industrie, plus pragmatique, sont souvent des applications simples, évitant la complexité et les incertitudes liées à l'intelligence artificielle. L'approche des laboratoires est plus théorique. Ceux-ci présentent des applications plus complexes et intelligentes, mais trop souvent incomplètement réalisées ou présentant des résultats non convaincants.

Plusieurs recherches (Gannoun, 2000; Emako-Lenou, 2000; Abu-Hakima, 1998; site Jorstad; Brewington, 1999; Bic, 1999) ont essayé de rapprocher ces deux concepts, mais peu donnent des résultats vraiment convaincants, tout en soutenant la technologie *agent-mobile*. La raison en est qu'il y a peu d'applications « standard » pour les agents mobiles et peu de mesures fiables. De plus, les caractéristiques qui avantagent les agents mobiles – souplesse, personnalisation – sont mal représentées par ces mesures. Dans le cadre de ce mémoire, nous allons chercher à voir dans quelle mesure il est possible, avec les technologies actuelles, de réconcilier les deux.

1.3 Objectifs de recherche

La recherche a pour but d'évaluer les performances et caractéristiques d'applications utilisant les technologies d'agents mobiles et d'intelligence artificielle. L'objectif principal de ce mémoire est de proposer une architecture qui intègre, de manière efficace, les concepts d'agents mobiles et d'agents intelligents. De manière plus spécifique, ce travail de recherche vise à :

- identifier les différentes catégories d'application dans lesquelles l'utilisation d'agents mobiles dotés d'une plus grande intelligence présente des potentialités intéressantes ;
- développer deux applications parmi celles précédemment identifiées qui utilisent des techniques d'IA ;

- concevoir une architecture capable de supporter le déploiement de telles applications;
- évaluer les avantages et les inconvénients de l'architecture proposée en regard de l'intelligence ajoutée aux agents et des méthodes conventionnelles.

1.4 Plan du mémoire

Le mémoire comprend cinq chapitres. Le chapitre II fait le point sur les technologies d'agents mobiles et de recherche d'information; il analyse également quelques résultats obtenus lors de précédentes expériences. Le chapitre III décrira l'architecture développée, les deux applications retenues ainsi que les algorithmes utilisés. Le chapitre IV présentera et analysera les résultats d'implémentation et de mise en oeuvre. En guise de conclusion, le chapitre V présentera une synthèse des travaux et esquissera des directions de recherche futures.

Chapitre II
Systèmes d'agents mobiles

Les systèmes d'agents appliqués en réseautique, de la gestion de réseau ou de flux à la recherche d'informations, ont gagné beaucoup d'intérêt avec l'adoption d'Internet, depuis les cinq dernières années. Ils offrent de grandes perspectives de recherche. Même s'il reste encore à prouver leur supériorité par rapport à l'architecture client/serveur toujours performante, ils auront un grand avenir dès que les problèmes qui limitent leurs capacités auront trouvé une solution acceptable. Ce chapitre se concentre sur les systèmes d'agents mobiles, leur caractérisation, leur architecture, leurs avantages et désavantages. Elle abordera également les agents de recherche d'information et les algorithmes qu'ils utilisent car ils sont repris dans les applications développées.

2.1 Caractérisation des agents mobiles

Les agents mobiles sont au carrefour de deux concepts plus anciens : les agents et la mobilité. Dans cette section, nous définissons ces deux concepts de base qui déterminent les caractéristiques des agents mobiles.

2.1.1 Agents

Le concept d'agent vient du domaine de l'intelligence artificielle (IA), à la fin des années 70. Un de leurs ancêtres étaient les « actors », introduits par Carl Hewitt (1977). Le concept d'agent lui-même est assez flou et a conduit à de multiples définitions. Un agent est communément défini comme une aide logicielle qui remplace l'utilisateur dans une tâche routinière et pénible (Gray, 1995), comme organiser l'horaire de réunions ou le tri du courrier (électronique), ou qui cherche et trie des informations correspondant aux intérêts de l'utilisateur (Hafner, 1995; Rogers, 1995). Cette définition a fait de "agent" un mot passe-partout dans les milieux académiques et commerciaux. Des

applications sont souvent dites construites sur des agents dans le seul but d'attirer l'attention ou d'augmenter les ventes. Cependant, ce terme est aussi utilisé à bon escient dans le domaine de l'intelligence artificielle, avec des définitions variées mais concordantes (Noriega, 1997).

En IA, un agent peut être vu comme un système, matériel ou logiciel, qui a une certaine autonomie sur son comportement, interagit avec des humains ou d'autres agents, perçoit et réagit à son environnement et a un comportement orienté vers un but. La notion d'agent fait donc référence à une personne qui agit dans un certain but et un certain contexte.

D'un point de vue légal, un agent est une personne qui agit selon un principe dans un but précis et sous délégation limitée d'autorité et de responsabilité. Wooldrige (1999) propose deux acceptions du concept d'agent. Une notion faible, dans laquelle un agent est caractérisé par trois propriétés: autonomie, sociabilité, réactivité, et mise en situation. Et une notion forte dans laquelle une délégation est faite à un système qui a un comportement réfléchi, voire émotionnel.

L'idée d'agent n'est donc pas nouvelle, mais a été limitée par les progrès en intelligence artificielle. Le développement d'Internet et l'explosion des informations qu'il contient a donné un nouvel essor à la recherche dans ce domaine.

2.1.2 Mobilité

Les systèmes d'agents mobiles héritent des techniques de migration de processus, qui visent à transférer un processus entre deux ordinateurs, un processus étant une abstraction contenant le code d'un programme, mais aussi ses données et son état d'exécution (Emako-Lenou, 2000). Ces techniques étaient habituellement implémentées au niveau du système d'exploitation. Leur principale difficulté était de transférer l'état d'un processus interne au système, ainsi que ses ressources (fichiers, drivers, ...). C'est pourquoi la migration de processus a d'abord été implémentée à l'aide de "canaux" par lesquels le processus communique avec son environnement (Powell et al., 1998).

Les systèmes basés sur des appels au noyau ont suivi (Douglis et al., 1998). Ces systèmes étaient des réussites de recherche, mais n'ont pas connu le succès commercial qu'ils méritent à cause du refus de la communauté des utilisateurs de partager leur ordinateur, et de leur complexité impliquant toujours le système d'exploitation. L'idée de migration de processus a donc failli principalement en raison du fait qu'elle requerrait un système d'exploitation complexe et exclusif. Toutefois, elle a introduit les notions de *code mobile* et de *mobilité.*

2.1.3 Caractéristiques des agents mobiles

Bien qu'un agent mobile soit défini comme une classe d'agent ayant comme caractéristique seconde la mobilité, il est plus approprié de le considérer comme l'aboutissement des abstractions mobiles, telles le code, les objets, ou les processus. De fait, les agents mobiles sont étudiés principalement par des laboratoires davantage liés au domaine des télécommunications qu'à celui de l'intelligence artificielle. Ils s'appuient peu sur des concepts d'IA, même si les recherches les plus récentes se concentrent sur ces aspects. Ils sont plutôt bâtis sur les notions de langages interprétés qui supportent le code mobile, l'indépendance par rapport aux systèmes d'exploitation et la mobilité objet.

2.2 Architecture des systèmes d'agents mobiles

Trois approches ont été utilisées pour la conception d'un système d'agents mobiles (SAM) (Karmouch et al., 1998). L'une consiste à utiliser un langage propriétaire dont les caractéristiques répondent aux exigences des SAM. Compaq™ a exploré cette voie avec le projet Obliq (Noriega, 1997). Une autre approche consiste à implémenter le SAM comme une extension d'un système d'exploitation (TACOMA, Hafner, 1995). Ces deux approches n'ont pas eu beaucoup de succès.

La dernière (et principale) approche consiste à construire le SAM comme une application particulière pouvant s'exécuter sur n'importe quel système d'exploitation. Ce système est en fait composé de deux parties: l'une fixe et installée sur les serveurs – la plate-forme - et les agents eux-mêmes. La plupart des systèmes ayant choisi cette

approche (Aglet, Concordia, Mole, Odyssey et Voyager, entre autres) se composent d'un ensemble de classes ajoutées à la Machine Virtuelle Java (MVJ). Les autres utilisent d'autres langages, souvent plus anciens que Java (D'Agent, Ara) (Gray, 1995). La plupart de ces derniers, devant l'énorme popularité de Java, ont intégré un interpréteur Java. D'autres ont même été entièrement reconstruits en Java, comme Telescript/Odyssey. Ces systèmes sont tous basés sur une architecture client/serveur et utilisent l'approche du "bac à sable" pour la sécurité de l'hôte. Nous considérons maintenant ces deux dernières architectures.

2.2.1 Architecture Java

Un SAM est construit sur la machine virtuelle Java (Lindholm et al., 1996), qui procure l'indépendance par rapport au système d'exploitation et une grande partie du support de communication et de réseau. Il assure également la sécurité de l'hôte avec le mécanisme du "bac à sable". La plupart des systèmes, comme Aglet, Concordia, ou Voyager utilisent la MVJ d'origine procurée par chaque navigateur Internet ou produit Java. D'autres, comme Sumatra, la modifient pour ajouter des fonctionnalités qui font défaut à la MVJ, comme la conscience du réseau (site ewatch), tout en gardant l'interface standard Java qui lui donne tout son intérêt. Les systèmes "classiques" essaient d'utiliser des classes Java ou des agents particuliers pour assurer ces fonctions. Cette architecture peut être représentée de façon générale par la Figure 2.1, les ellipses représentant des classes Java.

2.2.2 Architecture multi-langages

Les systèmes utilisant cette architecture essaient de surmonter les limitations de la machine virtuelle Java. Leur cœur est un "noyau" ou "serveur" implémentant les fonctions indépendantes du langage, comme le transport des agents, l'allocation des ressources, la sécurité, ou le séquençage des «threads». Les agents sont exécutés par l'interpréteur approprié à leur langage. L'avantage de cette approche est de supporter plusieurs langages de programmation, mais leur complexité et leur lenteur s'accroissent avec le nombre de langages supportés. La Figure 2.2 illustre cette architecture.

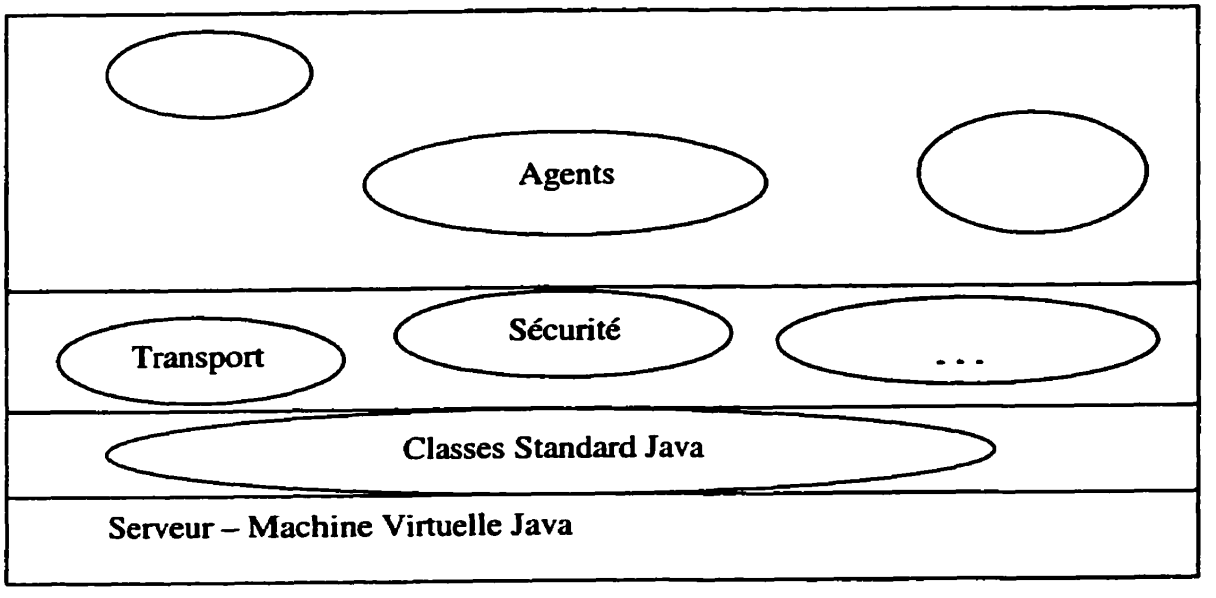


Figure 2.1 Architecture d'un SAM Java

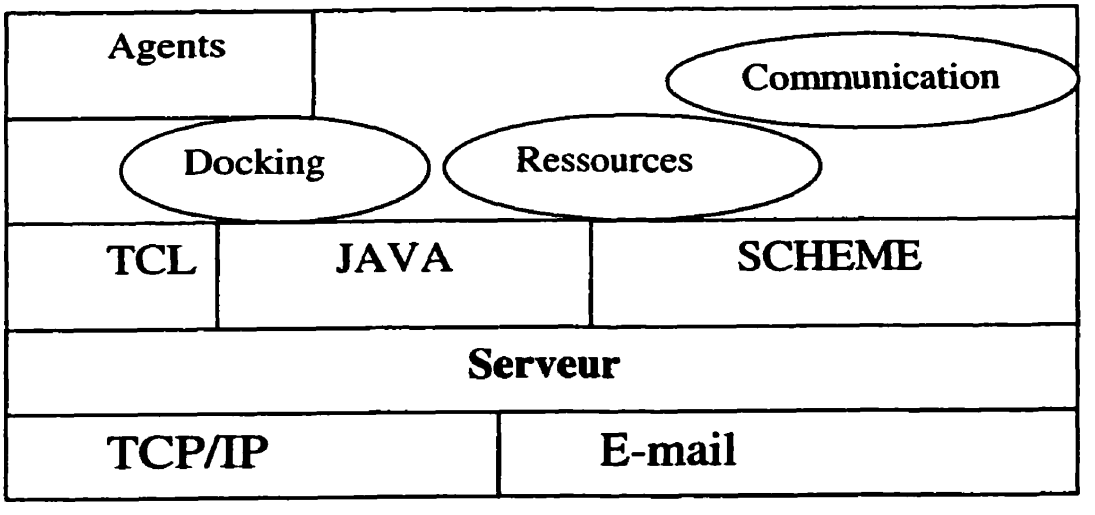


Figure 2.2 Architecture multi-langages (D'Agents)

2.2.3 Avantages et désavantages des agents mobiles

Selon Gray (1995), le récent intérêt pour les agents mobiles est alimenté par l'inadéquation croissante du modèle client/serveur traditionnel avec les applications réparties. En effet, dans une architecture client/serveur, le serveur procure un ensemble fixe d'opérations. Toutes les autres doivent être exécutées par le client. Si l'opération procurée par le serveur ne correspond pas exactement aux besoins du client, soit celui-ci doit effectuer plusieurs appels au serveur pour une seule opération, soit un programmeur doit l'ajouter sur le serveur. La première option ajoute des données intermédiaires sur le réseau, ce qui représente une perte de bande passante. La seconde option devient ingérable pour un grand nombre de clients. Les agents mobiles évitent ce gâchis de bande passante et permettent une exécution efficace, même quand le serveur ne procure pas d'opérations spécialisées, en migrant sur le serveur pour y effectuer tout calcul voulu avant de retourner le résultat final au client. Les agents qui font plus de travail évitent des messages intermédiaires et conservent plus de bande passante, ce qui les rend plus avantageux dans les réseaux à faible bande passante.

Cependant, si la taille du code de l'agent est trop grosse par rapport à la quantité de données accédées, cela peut affecter le gain de bande passante. Cela rend nécessaire un bon équilibre entre les capacités d'un agent et la complexité de la tâche qu'il a à accomplir.

Selon Lange (1998), les agents réduisent la charge du réseau. Les systèmes répartis reposent souvent sur des protocoles de communication qui impliquent de multiples interactions pour l'accomplissement d'une certaine tâche. C'est encore plus vrai quand il y a des mesures de sécurité. Les agents mobiles permettent d'encapsuler une conversation et de la rendre sur une destination hôte où les interactions vont se faire localement. Les agents mobiles sont aussi utiles pour réduire le flot de données brut sur le réseau. Quand un grand volume de données est stocké sur des serveurs distants, la manipulation de ces données devrait être faite localement, plutôt qu'en les transférant sur le réseau. Le principe est simple : amener le calcul aux données, plutôt que les

données au calcul. La Figure 2.3 montre comment les agents mobiles peuvent réduire la charge du réseau.

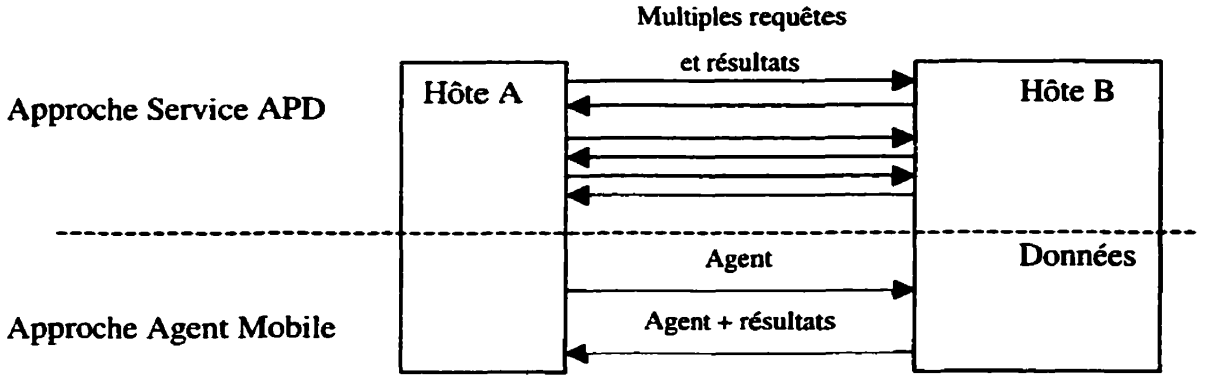


Figure 2.3 Réduction de la charge d'un réseau par les SAM

Les avantages et désavantages des agents mobiles peuvent être résumés comme suit :

Avantages attendus

- Extension des capacités de traitement en surmontant les limitations d'un petit ordinateur, comme un «palmtop». Il suffit d'envoyer un agent exécuter une tâche sur un serveur ayant de plus grandes capacités de calcul, de mémoire et de communication.

- Personnalisation : il est plus facile à un utilisateur de personnaliser son agent qu'un programme résidant sur un serveur distant.

- Survivabilité : alors qu'un programme classique est lié à une machine, un agent mobile peut se déplacer pour éviter une erreur matérielle ou logicielle, ou tout simplement un arrêt de la machine.

- Représentation d'un utilisateur déconnecté : un agent peut continuer à parcourir le web, même sans interaction avec l'utilisateur, grâce à son autonomie.

- Réduction de la charge du réseau : en se rapprochant des données sur lesquelles il veut travailler, un agent mobile peut permettre de diminuer le nombre de messages de communication, et par là même, la charge du réseau.

- Indépendance par rapport au système d'exploitation.

- Facilité de développement : le concept d'agent devrait (à terme) permettre de développer des applications mobiles plus facilement en masquant à l'utilisateur les problèmes liés au transport dans les réseaux, et même certains choix d'optimisation faisables par l'agent, ainsi qu'une analogie avec le monde réel. Un agent mobile et intelligent pourrait être vu comme un utilisateur humain habituel.

Désavantages

- Manque d'applications où les agents mobiles apportent un avantage certain.

- Sécurité : c'est une difficulté majeure pour le développement de systèmes d'agents mobiles, comme système fonctionnant en environnement ouvert (i.e. Internet) et non-fiable (réseaux mobiles).

- Manque d'infrastructure et de standards : les agents ont besoin du support fixe d'une « plate-forme ». Or, aucun standard n'est réellement appliqué et leur conception varie encore beaucoup d'un système à l'autre. De même, les difficultés non résolues empêchent la formation d'une infrastructure suffisamment éprouvée pour permettre le développement d'applications économiquement intéressantes.

En bref, même s'il existe un standard pour les systèmes multi-agents (MASIF, Miljicic, 1998), la technologie est trop peu mature dans beaucoup de domaines, dont celui de la sécurité, pour faire réellement sortir les agents des laboratoires.

2.3 Recherche d'informations

La recherche d'informations (RI) au sens large est sans doute le domaine d'application le plus fertile des agents. La raison en est qu'avec l'explosion du nombre de données disponibles sur les réseaux, les humains sont dépassés par la tâche de classer ces documents et de les retrouver suffisamment rapidement. On estime également que les moteurs de recherche fonctionnant sur des techniques de RI classiques n'arrivent qu'à archiver 15% du contenu d'Internet (Moussawi, 2000). On se trouve donc devant un problème où les techniques classiques sont dépassées et seuls des logiciels suffisamment autonomes et intelligents comme les agents peuvent donner des résultats dignes d'intérêt. Nous allons maintenant présenter les algorithmes de recherche d'information classiques ainsi que la technique d'apprentissage par retour d'information (feedback).

2.3.1 Techniques de recherche d'information automatisées

Les techniques de recherche d'information automatisées consistent à trouver, à partir d'une requête décrivant l'information cherchée, non pas cette information, mais un document la contenant. Le concept le plus important en recherche d'information est celui de correspondance. Un document correspond à une requête s'il contient l'information représentée par la requête. Cela pose le problème de représentation d'une information à plusieurs niveaux. La requête, telle que formulée en mots par l'utilisateur, ne représente pas exactement sa pensée; de plus, le système va souvent travailler sur une représentation interne de cette requête, ce qui introduit une nouvelle source d'erreur.

La première fonction de recherche automatique consistait à chercher le texte de la requête dans celui du document, ou un texte approchant. On en déterminait une distance, non nulle si le texte ne se retrouvait pas exactement dans le document. Devant la

complexité de cette méthode et ses mauvais résultats (en terme de correspondance), se sont développées des méthodes de recherche par mots, permettant de plus l'indexation des documents pour un retrait plus rapide des réponses.

On peut classer les moteurs de recherche actuels en quatre logiques de recherche (Moussawi, 2000) :

- Recherche géographique : ils permettent de chercher des sites par localisation géographique. Des exemples de cette classe sont : www.excite.com/travel et www.urec.cnrs.fr/annuaire .
- Recherche thématique : Dans ce type de recherche, les documents sont classés selon une structure d'arbre, par thèmes et sous-thèmes. Citons parmi les principaux : www.yahoo.com, magellan.excite.com, www.einet.net, www.nomade.fr .
- Recherche par index : c'est la principale catégorie de moteurs de recherche en raison de leur flexibilité. Les documents sont indexés, en général selon les mots qu'ils contiennent. La recherche peut ainsi se faire selon n'importe quels mots, sans se soucier s'ils ont été utilisés pour la classification des documents ou non.
- Recherche par méta-index : ces moteurs se contentent de reformuler la requête de l'utilisateur pour chacun des moteurs de recherche qu'ils connaissent et de classer ensuite les résultats : www.metacrawler.com, www.copernic.com, www.profusion.com, www.askjeeves.com.

Seules les techniques de recherche par index vont nous intéresser ici, en raison de leur plus grande flexibilité. On peut noter qu'aucune des catégories ci-dessus ne résout des problèmes tels que la véracité ou la pertinence de l'information trouvée. Pour remédier à certaines des limitations des moteurs de recherche, certaines expériences utilisent une structure multi-agent dans le cadre d'une « bibliothèque virtuelle ».

La plupart des stratégies de recherche d'information utilisent une fonction de correspondance qui mesure une distance entre un document et une requête ou un profil. Il existe de nombreuses distances, une des plus simples étant

$$M = \frac{2|D \cap Q|}{|D| + |Q|}$$

où D désigne l'ensemble des mots clé indexant le document et Q, celui représentant la requête. Elle calcule en fait le rapport des mots communs des deux documents au nombre total de mots.

Une des techniques les plus simples, mais aussi la plus utilisée dans les moteurs de recherche, est la recherche booléenne : un document est dit correspondre à une requête quand les mots de la requête se retrouvent dans le document. Elle est dite « booléenne » car les mots de la requête peuvent être groupés en expressions logiques booléennes (avec AND, OR, ...). Bien que ce soit une des techniques les plus simples, diverses méthodes sont utilisées pour la rendre plus efficace, comme les techniques d'expansion de requête, qui consistent à modifier la requête initiale, souvent en lui ajoutant des mots «synonymes». Son avantage est d'être rapide. Son défaut principal est de donner la même importance à chaque mot et à chaque document retrouvé (le document recherché peut aussi bien être classé 5^{e} ou 49^{e} sur 50 documents retrouvés).

2.3.2 Modèle espace-vectoriel

Pour pallier ce problème, Salton (1975) a introduit le modèle *espace-vectoriel*, basé sur la pondération des mots en fonction de leur importance dans un document ou une requête. Ce modèle est peu utilisé actuellement car ses performances sont fortement liées à la topologie de l'espace créé lors de l'indexation. Néanmoins, l'arrivée de systèmes de recherche évolués basés sur les agents remet ce modèle au devant de la scène.

Dans le modèle *espace-vectoriel*, le schéma figuratif de base est le vecteur (Salton et McGill, 1983). Ainsi, dans cette représentation, les documents et les requêtes sont considérés comme un ensemble de vecteurs dans un espace à n dimensions, n étant le nombre de mots utilisés dans l'indexation. Chaque document est représenté par un vecteur constitué des poids de chacun de ces mots dans la représentation du document. Ce poids peut aussi bien être un entier (ex : nombre d'occurrences du mot dans le document) qu'un nombre réel. La correspondance entre la requête et un document est

alors calculée comme l'inverse d'une distance entre les vecteurs de représentation respectifs dans cet espace de dimension n. Il existe plusieurs métriques possibles. Mathématiquement, toute fonction satisfaisant les trois propriétés de positivité, symétrie, et inégalité triangulaire, est une distance. Les métriques euclidiennes sont encore très populaires et utilisées en RI par Myaeng et Korfhage (1990), entre autres.

Une autre fonction populaire en RI est la corrélation cosinus. En supposant que le document et la requête sont représentés sous forme vectorielle, soit $Q = (q_1, q_2, \ldots, q_t)$ et $D = (d_1, d_2, \ldots, d_t)$ où q_i et d_i sont les poids associés à chaque mot clé i. La corrélation cosinus est simplement :

$$r = \frac{\sum_{i=1}^{t} q_i d_i}{\left(\sum_{i=1}^{t} (q_i)^2 \sum_{i=1}^{t} (d_i)^2 \right)^{1/2}}$$

Soit, dans un espace euclidien,

$$r = \frac{(Q,D)}{\|Q\| \|D\|} = \cos(\theta)$$

où θ est l'angle entre les vecteurs Q et D.

2.3.3 Algorithme d'apprentissage par rétroaction

Un utilisateur confronté à un système de recherche d'information automatique va sans doute vouloir utiliser une stratégie d'essais et de corrections plutôt qu'exprimer sa requête en une seule fois. Le genre d'informations dont il peut avoir besoin pour reformuler sa requête est:

(1) la fréquence d'occurrence de ses termes de recherche dans la base de documents ;

(2) le nombre de documents susceptibles de correspondre à sa requête ;

(3) des alternatives aux termes utilisés ;

(4) des citations susceptibles d'être trouvées ; et

(5) les termes utilisés dans les citations (4).

Un utilisateur peut alors préciser, élargir ou recentrer sa requête suivant les informations fournies par le système. C'est donc une forme manuelle de rétroaction. Souvent, une ou plusieurs de ces informations ne sont pas disponibles. Nous considérons maintenant une approche mathématique pour que le système puisse modifier automatiquement la requête. Le mot *rétroaction* est utilisé pour décrire le mécanisme par lequel un système peut améliorer ses performances en considérant son passé. Cette notion est bien établie en automatique et en RI et a été popularisée par Norbert Wiener dans son livre «Cybernetics» (1948).

Considérons une stratégie de recherche utilisant une fonction de correspondance M et des vecteurs t-dimensionnels pour les représentations de la requête Q et du document D. On suppose ici que l'ensemble des documents est connu. Le but est de retrouver les documents voulus A sans les documents indésirables A'. Malheureusement, la correspondance est une notion propre à l'interprétation de l'utilisateur et celle-ci peut différer de la description qu'il en donne au système. Dans le cas où M est la fonction de corrélation cosinus, la procédure de décision avec un seuil T (qui décide de montrer un document ou non), $M(Q,D) - T > 0$, correspond à une fonction de discriminant linéaire utilisée pour séparer deux ensembles A et A' dans R[t]. Nilsson (1965) a expliqué comment ces fonctions peuvent être entraînées en modifiant les poids q_i. Supposons que A et A' soient connus, alors la formulation correcte de la requête Q_0 est celle pour laquelle

$$M(Q_0,D) > T \text{ quand } D \in A \text{ et}$$
$$M(Q_0,D) \leq T \text{ quand } D \in A'.$$

Un théorème (Nilsson, 1965) assure que, pour tout Q et Q_0, il existe une procédure itérative faisant converger Q vers Q_0, appelée la procédure de correction par incrément fixe. Elle s'énonce ainsi:

$$Q_i = Q_{i-1} + cD \text{ si } M(Q_{i-1}, D) - T \leq 0 \text{ si } D \in A$$
$$Q_i = Q_{i-1} - cD \text{ si } M(Q_{i-1}, D) - T > 0 \text{ si } D \in A'$$

et aucun changement ne se produit si le diagnostic est correct. c est l'incrément de correction, de valeur arbitraire et habituellement égale à 1. En pratique, il peut être nécessaire de répéter l'opération un grand nombre de fois avant convergence.

La situation est moins simple en RI car les ensembles A et A' ne sont pas connus à l'avance, mais on peut demander à l'utilisateur de décider lesquels sont désirés parmi les documents trouvés. Le système peut alors modifier Q automatiquement. Dans sa thèse, Rocchio (1966) a défini la requête optimale comme celle qui maximise:

$$\Phi=\frac{1}{|A|}\sum M(Q,D)-\frac{1}{|\bar{A}|}\sum M(Q,D)$$

Si M est la fonction cosinus, il est facile de montrer que Φ est maximisé par :

$$Q_0 = c\left(\frac{1}{|A|}\sum_{D\in A}\frac{D}{\|D\|}-\frac{1}{|\bar{A}|}\sum_{D\in A}\frac{D}{\|D\|}\right)$$

où c est une constante proportionnelle arbitraire. Si les sommes ne portent que sur les documents trouvés à l'itération i, on obtient une requête optimale pour ceux-ci, mais elle peut ne pas l'être pour les documents non encore retournés. On ajoute alors ce vecteur à la précédente formulation de la requête pour obtenir:

$$Q_{i+1} = w_1 Q_i + w_2\left(\frac{1}{|A\cap D_i|}\sum_{D\in A\cap D_i}\frac{D}{\|D\|}-\frac{1}{|\bar{A}\cap D_i|}\sum_{D\in \bar{A}\cap D_i}\frac{D}{\|D\|}\right)$$

où w_1 et w_2 sont des coefficients de pondération.

Salton (1975) a en fait utilisé une version légèrement modifiée. En résumé, ces ajustements consistent à donner plus de poids dans la description de la requête aux mots qui décrivent les documents voulus et moins aux autres. Les essais ont démontré que cette technique peut être très efficace, mais oblige l'utilisateur à juger un certain nombre de documents à chaque requête. Les systèmes multi-agents cherchent à éviter ce défaut en faisant partager la tâche entre un grand nombre d'utilisateurs humains ou agents logiciels collaborant.

2.4 Applications des systèmes d'agents mobiles

Les domaines d'application des SAM sont à peu près les mêmes que ceux où les agents statiques sont actuellement appliqués : recherche et filtrage d'informations, commerce électronique et télécommunications.

2.4.1 Recherche et filtrage d'informations

C'est le plus grand champ d'application et d'expérimentation des agents. Même si les systèmes décrits ici utilisent des agents statiques, ils pourraient être avantageusement remplacés ou secondés par des SAM. Les premiers systèmes commerciaux utilisant des agents étaient des agents "moniteurs". C'étaient des programmes qui alertaient l'utilisateur quand une information intéressante apparaît. e-Watch, ZDNet (qui a cessé récemment ce projet) et Excite (Sites Internet) procurent ce service pour les nouvelles et les informations.

Alexa (Site Alexa) est une barre d'outils d'aide gratuite à la navigation. Elle procure des informations statistiques et des liens sur chaque site visité. Elle aide également au magasinage en ligne en vérifiant l'identité d'un site de commerce à partir des contacts donnés sur la page.

Les agents, et spécialement les agents mobiles, sont adaptés pour agir comme des "bots", des logiciels qui naviguent continuellement sur la toile pour trouver de nouvelles informations. Cette technologie, qui utilise des techniques de data-mining, est déjà bien étudiée et possède ses standards.

2.4.2 Commerce électronique

Les agents, fixes ou mobiles, peuvent être utilisés pour le commerce électronique de plusieurs façons. Ils peuvent donner aux utilisateurs un accès personnalisé aux informations données en ligne. Frictionless (Site Frictionless) est un système d'agents statiques intelligents, développé au MIT. Il permet aux utilisateurs de comparer plusieurs produits et conditions de ventes, en magasinant en ligne. L'utilisateur peut sélectionner un profil qui correspond à ses habitudes d'achat, choisir un produit et préciser ses caractéristiques. Ils peuvent aussi être utilisés pour des ventes aux enchères,

comme AuctionBot, développé à l'Université de Michigan. Il existe beaucoup d'autres sites d'enchères utilisant des agents statiques (ebay, onsale, auctionet).

Parmi les systèmes d'agents mobiles, on peut remarquer Nomad (2000), utilisé dans un site d'enchères, eAuctionHouse, à l'Université de Washington. Tabican (site Aglets, détails en Japonais) est un marché virtuel pour des billets d'avions et des tours (air + hôtel) où des milliers d'agents mobiles de clients et vendeurs peuvent se rencontrer et trouver le meilleur prix pour le client et le vendeur à la fois, sans leur faire perdre de temps.

2.4.3 Agents mobiles en télécommunications

La fonctionnalité de mobilité prend toute son importance dans le domaine des télécommunications. Les agents mobiles peuvent y être utilisés pour couvrir toutes les couches des protocoles de communication, de la maintenance de réseau, jusqu'aux applications mobiles, suivant l'usager dans ses déplacements.

Dans le système SPIN (Abu-Hakima et al., 1998), un "Personal Communicator Agent™" (PCA) est un agent mobile chargé de délivrer un message au destinataire, quel que soit son appareil – téléavertisseur, téléphone, ordinateur, portable ou téléphone sans fil. Le PCA d'un utilisateur doit pouvoir recevoir les messages et les conduire sans interruption dans des réseaux hétérogènes à l'utilisateur. Par exemple, si le seul moyen de délivrer un message urgent à un utilisateur est un téléphone sans fil, l'agent personnel doit convertir le message textuel en message vocal.

NetChaser (di Stefano et C. Santoro, 2000) est un ensemble d'assistants personnels développé à l'université de Catania, en utilisant leur propre système d'agents mobiles. La mobilité permet à ces agents de suivre l'utilisateur même quand il change de machine.

Le domaine de l'administration de réseau est également l'objet de nombreuses recherches. Comme l'équipe d'IBM (site Recherche-IBM), beaucoup pensent que le

futur des réseaux réside dans une plus grande intelligence, pour plus d'adaptabilité et de mobilité, et que cette évolution passe par les agents mobiles.

L'administration de réseau est par nature asynchrone et répartie. De plus, il est souvent important d'avoir une vue locale du système pour pouvoir déterminer les causes et conséquences d'un problème. L'administrateur doit alors se déplacer vers des machines lointaines qui nécessitent des tâches de maintenance ou de mise à jour. L'installation et la maintenance des logiciels deviennent difficile avec l'augmentation du nombre de machines. Les agents mobiles sont ici adéquats pour voyager dans le réseau et effectuer des tâches périodiques.

2.5 Quelques systèmes existants

Bien que de nombreuses applications basées sur les agents aient été développées, peu ont passé le stade expérimental et encore moins celui de la commercialisation. On peut trouver une liste plus complète de SAM à (site liste). Nous allons citer ici seulement les plus importantes en tests, publications et applications.

Les *Aglets* d'IBM (site Aglets) ont été le premier système commercial développé en Java à IBM Tokyo Research Laboratory. *Concordia* (site Concordia) est un système commercial développé en Java par Mitsubishi Electric ITA. *Mole* (site Mole), développé à l'université de Stuttgart, implémente le schéma de migration faible, où sont transportées seulement les informations des données, car la migration forte (transport de tout l'état système de l'agent) était trop coûteuse. Quant à *Sumatra* (site Sumatra), il a été développé pour mesurer la performance des agents en gestion de réseaux. Il utilise une application, *Komodo*, qui gère le temps de réponse du réseau. L'application de test est *Adaptalk*, un logiciel de discussion sur Internet qui se positionne automatiquement à l'endroit optimal sur le réseau.

Voyager (site ObjectSpace) est un "Object Request Broker" (ORB) en Java supportant les agents mobiles. Malgré ses possibilités avancées de communication, ce n'est pas un système conçu spécifiquement autour des agents mobiles.

Agent-Tcl (site D'Agents) est l'un des premiers systèmes d'agents mobiles (première publication en 1995), initialement écrit en Tcl, un langage portable conçu au Dartmouth College. Il a été depuis réécrit pour supporter divers langages - Tcl, Java et Scheme – et renommé *D'Agents*.

Grasshopper est un système d'agents mobiles en pleine expansion développé à l'IKV, en Allemagne. Il présente des fonctionnalités intéressantes comme une interface graphique élaborée. Il est conçu entièrement en Java.

Une difficulté dans le développement de SAM est le manque d'environnements de test "mondiaux", où les agents peuvent traverser de grandes distances sur des réseaux de caractéristiques et capacités diverses. L'Université de Dartmouth et plusieurs autres institutions (site ActComm) se sont réunies pour construire un tel environnement. Chaque institution fournit une machine 386/486/Pentium tournant sous Linux, avec un compte pour chacune des autres institutions participantes. Chacun peut alors installer son système pour des tests et analyses de performance. Les participants à ce projet sont : Dartmouth, avec D'Agents, Berkeley et l'Université de Genève avec Messengers, Aachen University of technology et CNRI avec KnowBots. Malheureusement, ce projet a été abandonné (voir page Dartmouth-réseau).

Le "Institute for Information Technology" (IIT) du Conseil national de la recherche du Canada (CNRC) utilise un environnement réel (Emako-Lenou, 2000; Abu-Hakima et al., 1998) pour deux applications complémentaires de "seamless personal information networking" (SPIN™). Nous ne décrivons ici que la première: *seamless messaging* (SM); l'autre est un gestionnaire de réseau intelligent. Le réseau comprend un LAN de plus de 30 terminaux, une passerelle SS7, un serveur "Computer Telephony Integration" (CTI), un accès au LAN sans fil, une station de base sans fil, une passerelle ATM et de nombreux téléphones et portables. Une plate-forme CTI permet aux utilisateurs de recevoir des appels avec les informations correspondantes sur leur

terminal. Des "Diagnostic Agent™" (DA) sont déployés sur chaque nœud du réseau pour gérer chacun un type de matériel.

2.6 Performances des SAM et perspectives

"Messengers" (site Messenger) est un système développé à l'Université de California à Irvine. Il se compose de "messengers", des agents mobiles voyageant sur un réseau virtuel pouvant se superposer à un réseau réel. Ils peuvent suivre un programme, selon un "temps virtuel", et contenir du code natif.

Les Messengers ont été évalués sur un réseau de stations SunSPARC ELC's (16 MB mémoire chaque) connectées par un réseau Ethernet à 10 Mbps. L'interpréteur est un unique processus Unix, avec deux «threads» concurrents (un pour les envois, l'autre pour les réceptions) pour la communication avec les autres sur le réseau. Les tests montrent que les opérations arithmétiques sur les scripts Messengers ajoutent un surcoût d'un certain ordre de magnitude par rapport au code natif. L'appel de fonction dans Messengers est 100 fois plus coûteuse qu'un appel de fonction C. La création d'un messenger demande environ trois fois plus de temps que de créer un «thread» lwp (Low Weight Process), mais moins qu'un processus Unix. Finalement, les changements de contexte sont comparables pour chaque système.

Ceci montre que l'interprétation des scripts Messengers peut augmenter considérablement le coût. Cependant, ce coût ne se retrouve pas nécessairement dans les applications. La raison en est que les messengers peuvent contenir également du code natif C. De plus, la communication est la source la plus courante de surcoût dans les applications réparties, et l'utilisation des messengers réduit ce coût.

Les résultats montrent que les «threads» lwp et les messengers sont tous deux plus rapides que l'exécution séquentielle au delà d'une certaine granularité (1000 – 10000 opérations). Le surcoût des messengers devient même négligeable comparé à l'exécution «multi-thread» conventionnelle.

D'autres évaluations sont reprises dans (Bic et al., 1999), où un calcul (d'une image de la fractale de Mandelbrott) est exécuté comparativement avec les agents Messengers, un système réparti fonctionnant par passage de messages, et un programme C fixe. Le résultat montre un avantage des Messengers et du système réparti sur le programme C. La comparaison entre les deux premiers dépend de la granularité du calcul (le nombre de points calculés par chaque unité de calcul), et aucun des systèmes ne montre une réelle supériorité.

Selon David Kotz et Robert S. Gray (1999), de Dartmouth, les systèmes d'agents mobiles épargnent des délais dus au réseau et de la bande passante, au détriment de la charge de calcul sur les serveurs, car les agents sont souvent écrits dans des langages interprétés relativement lents. Ainsi, en l'absence de déconnexion réseau, les agents mobiles (surtout ceux qui doivent effectuer seulement quelques opérations par serveur) prennent souvent plus de temps que des implémentations traditionnelles pour accomplir une certaine tâche, car le gain en temps sur le trafic réseau est alors insuffisant pour compenser la lenteur d'exécution et de migration.

Heureusement, des progrès significatifs ont été faits dans le domaine de la compilation à l'exécution (surtout pour Java), l'isolation des fautes, et d'autres techniques (Muller et al., 1997), ce qui permet au code mobile de s'exécuter presque aussi vite que du code compilé. De plus, des groupes de recherche travaillent activement à réduire le coût de migration. Réunis, ces efforts devraient conduire à des systèmes dans lesquels l'utilisation d'agents mobiles n'impliquerait qu'une faible charge supplémentaire au serveur par rapport à un service fourni comme une procédure.

Par ailleurs, le projet ActComm (site ActComm) - transportable agents for wireless communications – a donné lieu à plusieurs évaluations. Dans (Brewington et al., 1999), on décrit un système de recherche d'information utilisant SMART et des agents mobiles. Les requêtes sont envoyées d'un laptop à 200 MHz à un serveur Sun via une connexion Ethernet à 10 Mbps. Les agents utilisent la nouvelle version «multi-thread» de D'Agents, sans encryption. Les résultats montrent que la migration implique un coût supplémentaire par rapport à la communication inter-agent (deux agents fixes

communicant) et encore plus qu'un APD. Les chercheurs identifient deux raisons principales à ces mauvaises performances : TCL est lent car trop de code doit encore être exécuté par l'interpréteur avant de recevoir un nouvel agent; et l'utilisation de TCP, plus lent qu'UDP pour la communication et l'envoi des agents. Mais les agents surpassent APD quand le nombre de requêtes augmente, sauf si les documents doivent être retournés à l'utilisateur par le réseau pour vérification.

Les résultats peuvent ne pas sembler très concluants, mais il faut considérer aussi que même avec un lien de 10 Mbps (rapide) et un interpréteur Tcl lent, le SAM surpasse la solution client/serveur dans plusieurs cas, en particulier quand les opérations de recherche sont fournies par le serveur sous forme d'une librairie. Dans tous les cas, l'utilisation d'agents mobiles sauve de la bande passante réseau.

Jorstad (site Jorstad) donne un exemple classique de comparaison entre un système classique client/serveur et un SAM. Le scénario (liaison à faible débit, serveurs de bases de données distants) est très avantageux pour l'agent, qui surpasse l'autre système en charge de réseau et en rapidité.

Domain Name eXchange (DNX) (Gannoun, 2000; site DNX) est un système développé à l'université de Genève. Son but est de fournir un SAM qui aiderait à gérer la demande croissante pour les noms de domaine, qui ont désormais valeur commerciale. Ce système utilise à la fois des agents fixes et mobiles, sur la plate-forme *JavaSeal*, développée dans la même université. Cette plate-forme peut supporter plus de 1,100 agents concurrents, ayant chacun un «thread» d'exécution actif. Au delà, des créations supplémentaires sont très lentes du fait de la quantité importante de basculement générée. La communication inter-agents ralentit également. Ils prévoient d'améliorer les performances de la plate-forme en y incluant de nouvelles fonctionnalités comme le stockage d'agents inactifs.

La performance en termes de rapidité et de bande passante utilisée n'est pas le seul avantage des SAM et n'est pas toujours le facteur déterminant du succès. Todd Papaioannou et John Edwards, de l'université Loughborough, UK, proposent différentes mesures de performance dans (site Loughborough). Les deux systèmes décrits sont

construits pour supporter les ventes d'une entreprise de fabrication répartie, en utilisant le "Aglets Software Development Kit" d'IBM. Les bases de l'implémentation proviennent de données collectées dans l'étude d'un cas réel. Les deux systèmes sont évalués en utilisant la méthodologie «Goal/Question/Metric» (but/question/métrique). Deux nouvelles mesures d'alignement sémantique et de capacité de changement sont présentées et utilisées pour l'évaluation des systèmes. Ceux-ci sont évalués à partir d'un ensemble de scénarios générés lors de l'étude de cas. Ensuite sont examinées les implications de l'utilisation de code mobile, comparativement à une technologie répartie traditionnelle. Ce travail met en évidence que les systèmes d'agents mobiles et d'objets mobiles ont des propriétés inhérentes qui peuvent être utilisées pour la construction de systèmes répartis adaptables. L'autonomie des agents mobiles donne encore plus de support.

On peut considérer que les tests choisis sont suffisamment récents (1998, 99) pour refléter les performances actuelles des systèmes d'agents mobiles. Les évaluations décrites sont similaires, dans le fait qu'elles sont relativement simples, impliquant un faible nombre de nœuds de réseau et des conditions de tests favorables aux SAM. En résumé, les résultats obtenus indiquent que les SAM utilisent moins de bande passante réseau (Gray, 1999; Brewington et al., 1999; Gannoun et al., 2000; site Jorstad), mais ils arrivent encore difficilement au niveau des systèmes traditionnels pour la rapidité d'exécution (Brewington et al., 1999) (les scénarios de test avantagent souvent les agents), et encore moins pour la charge de calcul des serveurs (Gray, 1999). Les SAM conviennent donc aux applications où l'optimisation de la charge du réseau est plus importante que la rapidité, comme c'est le cas de la plupart des applications en environnement sans-fil.

On peut se demander, au vu de ces résultats, si des applications de SAM vont être commercialisées dans un proche avenir ou s'ils vont perdre la considération (et les fonds de recherche) des entreprises, comme l'IA dans les années 80. Mais les agents mobiles ont l'avantage de permettre à la fois le développement rapide d'applications et la réutilisation des travaux faits en IA au cours des 20 dernières années. La mobilité leur

confère également d'autres avantages, comme la facilité de personnalisation, l'adaptabilité ou l'interopérabilité.

Certaines recherches essaient de mettre ces qualités en avant en évaluant des paramètres autres que la seule performance (site Loughborough), comme la sûreté ou l'adaptabilité, montrant ainsi les vrais avantages des SAM sur les systèmes traditionnels. De plus, certains articles montrent que la technologie agent a suffisamment évolué pour permettre la gestion d'un réseau (Abu-Hakima, 1998), et le développement d'environnements de test réalistes (Abu-Hakima, 1998; Site Dartmouth-réseau) nous laisse présager de plus en plus d'évaluations et de comparaisons dans un futur proche.

Les systèmes d'agents mobiles permettent le développement rapide d'applications qui utilisent beaucoup la mobilité et peu d'algorithmes complexes d'intelligence artificielle. Ces systèmes montrent de bonnes performances sur des réseaux fermés et sûrs, mais ils ont encore besoin de plus d'autonomie et d'intelligence pour aborder des réseaux plus changeants et risqués. Or, au vu de leurs caractéristiques, les systèmes d'agents mobiles sont faits pour ces derniers réseaux. Il y a donc là une contradiction flagrante entre les capacités actuelles des agents mobiles et ce pour quoi ils sont prévus et adaptés. C'est pourquoi il est important d'aborder des problèmes tels que le "travelling agent" (Brewington, 1999), ou reroutage (Jorstad).

L'intégration dans les agents mobiles d'algorithmes complexes d'IA augmente la taille de leur code et les rend moins efficaces pour des tâches simples, mais cela leur permet d'effectuer des tâches plus complexes dans des environnements plus changeants et hasardeux. L'important est de ne pas surestimer les agents mobiles et de garder un bon équilibre entre la difficulté de la tâche à accomplir et la quantité d'IA inclue, lors du développement d'un système (Woolridge, 1999).

Les tests et évaluations décrites ici ont pu montrer la supériorité d'un système client/serveur ou d'un SAM, suivant la situation. En fait, les SAM semblent surpasser les solutions classiques pour la charge du réseau ou même la vitesse d'exécution, mais les vrais tests vont à peine commencer, avec les environnements réalistes comme SPIN (Abu-Hakima, 1998) ou (Dartmouth-réseau), et des applications plus complexes.

Les agents mobiles sont l'aboutissement de l'évolution des concepts de mobilité, mais ils sont aussi « 99% computing, 1% AI » (Woolridge., 1999), et ce 1% doit prendre de plus en plus d'importance pour pouvoir exploiter toutes les possibilités que l'on peut attendre des agents mobiles.

Chapitre III
Architecture multi-agents de recherche d'information

On peut classer les applications utilisant des agents mobiles en trois grandes catégories : une où l'agent est très spécialisé et va effectuer une tâche simple sur un ou plusieurs serveurs; une autre où l'agent effectue une tâche complexe sur une longue durée et se déplace peu; dans la dernière, l'agent doit effectuer une tâche complexe sur un certain nombre de serveurs. Cette dernière catégorie comprend la plupart des applications de commerce électronique et de recherche d'information. La rapidité d'exécution et la taille du code sont alors des facteurs cruciaux pour l'agent, alors qu'il doit également pouvoir effectuer des tâches complexes et, pour cela, avoir accès au code d'algorithmes pouvant être complexe et requérant souvent un grand volume de données. Or, ces algorithmes et ces données gagneraient à être partagés avec d'autres applications ou être fournis par le serveur lui-même, pour être accédés localement. Il est également inutile que l'agent transporte du code déjà présent ou effectuant une opération déjà fournie par le serveur. Ce chapitre traite de la conception d'une architecture multi-agents résolvant ce problème et destinée en particulier aux applications de recherche d'information, mais pouvant être étendue à d'autres types d'applications.

3.1 Caractérisation de l'architecture

Considérant les limitations d'une application basée sur un agent mobile seul (transport de tout le code à chaque déplacement), nous proposons de scinder l'agent unique en plusieurs agents intégrés dans une architecture multi-agents, la plupart d'entre eux toujours mobiles. Nous allons ici décrire cette architecture, en commençant par définir ses objectifs et spécifications.

Les qualités les plus fréquemment attribuées aux agents mobiles sont de diminuer la charge d'un réseau et d'avoir une «conscience» du réseau, c'est-à-dire de sa topologie et de sa configuration. Bien que souvent prises pour acquises, ces caractéristiques sont

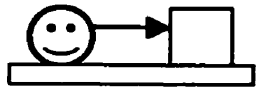
en fait rarement implantées dans les systèmes d'agents mobiles actuels. De plus, l'approche classique cherche au contraire à masquer au maximum les caractéristiques physiques du réseau aux applications par des couches successives de protocoles. Nous allons donc chercher à construire notre architecture de façon à minimiser la charge du réseau. En particulier, l'un des grands défauts de l'approche client/serveur et des réseaux classiques en général est de ne procurer aucun moyen de localisation géographique et d'encourager du gaspillage de bande passante en traitant de la même façon une liaison locale et une liaison intercontinentale pouvant comporter un grand nombre de « hops ». Ces dernières sont souvent surchargées aux heures de pointe et peuvent entraîner des délais importants dans l'acheminement des paquets. Cela tend à changer, en particulier dans le domaine fortement concurrentiel WAP de la téléphonie cellulaire, mais beaucoup reste à faire. Nous allons donc mettre l'accent sur les aspects de localité et de réutilisation du code.

3.1.1 Catégories d'agents

Comme Esmahi (1999), on va distinguer essentiellement deux types d'agents, les agents passifs ou réactifs, et les agents actifs. Pour résumer, les premiers ne vont agir qu'en réponse à un message de leur environnement (utilisateur, système ou autre agent) alors que les seconds vont agir de leur propre initiative. Ils peuvent déclencher une action à la suite d'un événement interne, même sans message extérieur. Généralement, les agents actifs vont être les acteurs principaux et vont utiliser les agents passifs pour accomplir la tâche qui leur a été confiée. Toutefois, contrairement à des objets, les agents passifs sont permanents et conservent un état interne qui conditionne leur réponse aux messages qu'ils reçoivent. Par conséquent, leur réponse à deux messages identiques peut varier selon le passé de l'agent. C'est ce qui distingue la programmation orientée agent, introduite par Shoham (1993), de la programmation orientée objet. Les agents peuvent très bien passer d'un état passif à un état actif ou inversement à tout moment, à la réception d'un message particulier ou à un moment donné. Un agent actif devient passif lorsqu'il se met à la disposition des agents actifs en attente du prochain événement

(qui peut être interne à l'agent). Par exemple, un agent d'analyse financière peut partager ses fonctions d'analyse avec d'autres agents pendant qu'il attend des données, comme le montre la Figure 3.1.

L'agent mobile actif arrive sur le serveur de données boursières.



Il demande au serveur de lui donner le prochain passage d'une action au-dessus d'un certain seuil et devient passif en attendant la réponse.

D'autres agents peuvent lui demander d'analyser le cours d'une autre action pendant ce temps.

Le serveur lui fournit les renseignements attendus et l'agent redevient actif pour continuer sa tâche (par exemple trouver comment rejoindre son propriétaire le plus rapidement possible).

Figure 3.1 Agents actifs et passifs

Un agent «multi-thread» peut être en même temps actif et passif. Dans notre cas, les agents actifs sont mobiles et, pour rester suffisamment légers, ne peuvent pas emporter une grande quantité de connaissances sur les autres agents. Un des objectifs de l'architecture décrite ici est donc de procurer à ces agents un moyen simple de trouver un agent capable de fournir le service dont ils ont besoin. Tout d'abord, comment désigner un «service»? Nous avons choisi les interfaces comme la caractéristique d'un

agent représentant une fonction ou un ensemble de fonctions qu'un agent est capable de réaliser pour d'autres, ceci pour plusieurs raisons. La caractéristique choisie doit être aussi représentative que possible de la fonction de l'agent et doit être indépendante d'autres paramètres. Il doit être possible et aisé de chercher un agent selon cette caractéristique. Enfin, un agent doit pouvoir regrouper plusieurs fonctionnalités. Grasshopper, sous lequel nous avons développé les applications, présentait plusieurs possibilités de recherche, selon le nom, la description, ou la classe de l'agent. Cependant, ces caractéristiques sont soit uniques, comme le nom ou la classe, soit trop vagues et changeantes, comme la description. Or, il est naturel, en Java ou tout autre langage orienté objet, de penser à une interface pour représenter un ensemble de fonctions qu'un objet contient. La connaissance de cette interface est même nécessaire pour pouvoir communiquer avec un autre agent. Plutôt que d'imposer une interface standard unique, nous avons préféré choisir l'interface comme représentative d'un service offert par un agent. La fonction de recherche correspondante, inexistante sous Grasshopper, est donc réalisée par notre architecture (Figure 3.2) au niveau du registraire, agent particulier décrit dans le paragraphe suivant.

Le registraire est un agent particulier de notre architecture (le seul), car il concentre les fonctions qui nous sont nécessaires mais qui ne sont pas assurées par la plate-forme. Nous avons vu que l'une de ces fonctions est la recherche d'autres agents. Pour cela, le registraire va conserver une liste des agents présents sur la plate-forme avec leurs interfaces. Les agents passifs vont, à leur arrivée sur la machine, s'inscrire auprès du registraire pour chaque interface qu'ils désirent présenter. Ils se désinscrivent lors de leur départ ou de leur suppression. Le registraire peut être informé par la plate-forme de l'arrivée ou du départ d'un agent, mais c'est aux agents eux-mêmes de décider selon quelles interfaces ils désirent pouvoir être recherchés. Le processus d'inscription ne peut donc pas être complètement automatisé et ignoré au niveau du code des agents.

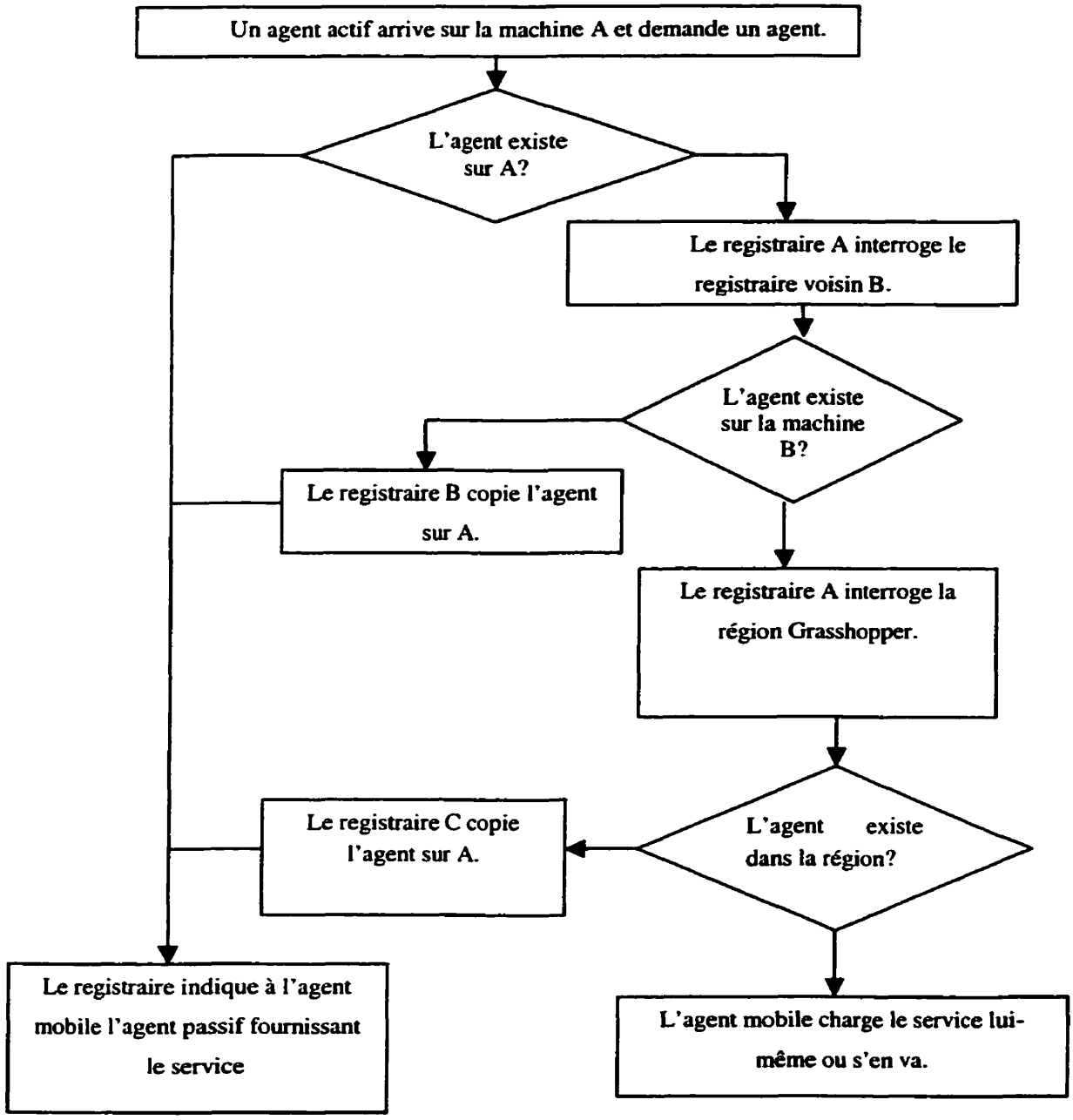


Figure 3.2 Algorithme de recherche d'un agent

Si le registraire ne trouve pas un agent dans sa propre base, il interroge les registraires «voisins» en vue de tirer profit de la localité pour réduire la charge du réseau (nous verrons ensuite comment est transcrite cette idée), puis l'ensemble de la région de

sécurité à laquelle il appartient. En cas de succès, il copie l'agent trouvé sur la machine locale et lui transmet la requête de l'agent mobile. Ainsi, on n'utilise qu'en dernier recours les liaisons et les machines « distantes », c'est à dire n'appartenant pas au même sous-réseau. On économise donc des ressources réseau. On peut noter qu'un avantage de ce choix est aussi de libérer l'agent mobile du code préparant et effectuant la recherche, ainsi que du code de traitement d'erreurs associé qui peut représenter une part importante de la taille d'un agent. De plus, ce code variant selon le système d'agents mobiles utilisés, on acquiert une plus grande indépendance au système utilisé lors du développement de l'agent. La Figure 3.3 illustre le déroulement d'une communication entre un agent mobile actif et un agent passif par un exemple tiré des applications développées.

En vue de minimiser la charge totale du réseau et de rendre les applications plus performantes, il nous faut introduire un élément de « localisation » dans le système. Plusieurs approches sont possibles. Grasshopper procure déjà une certaine «régionalisation» implicite de l'espace avec les «régions». Celles-ci réunissent les agences de même caractéristiques de sécurité et de propriété, ce qui indique une certaine proximité géographique. Elles procurent également des fonctions de recherche d'agents. Cela n'étant pas suffisant (un réseau d'entreprise internationale pourrait s'étendre sur plusieurs pays avec la même politique de sécurité), le registraire sera donc chargé de conserver et de procurer aux autres agents des indications topologiques sur le réseau, en particulier, les agences et les agents «proches». Cette notion de proximité peut très bien varier d'un registraire à l'autre. Elle reflète la politique du propriétaire du système en matière d'utilisation du réseau. Des «voisinages» plus grands impliquent en effet une plus grande utilisation du réseau local et pourraient être préférés par un administrateur ayant des liaisons haut-débit sous-utilisées.

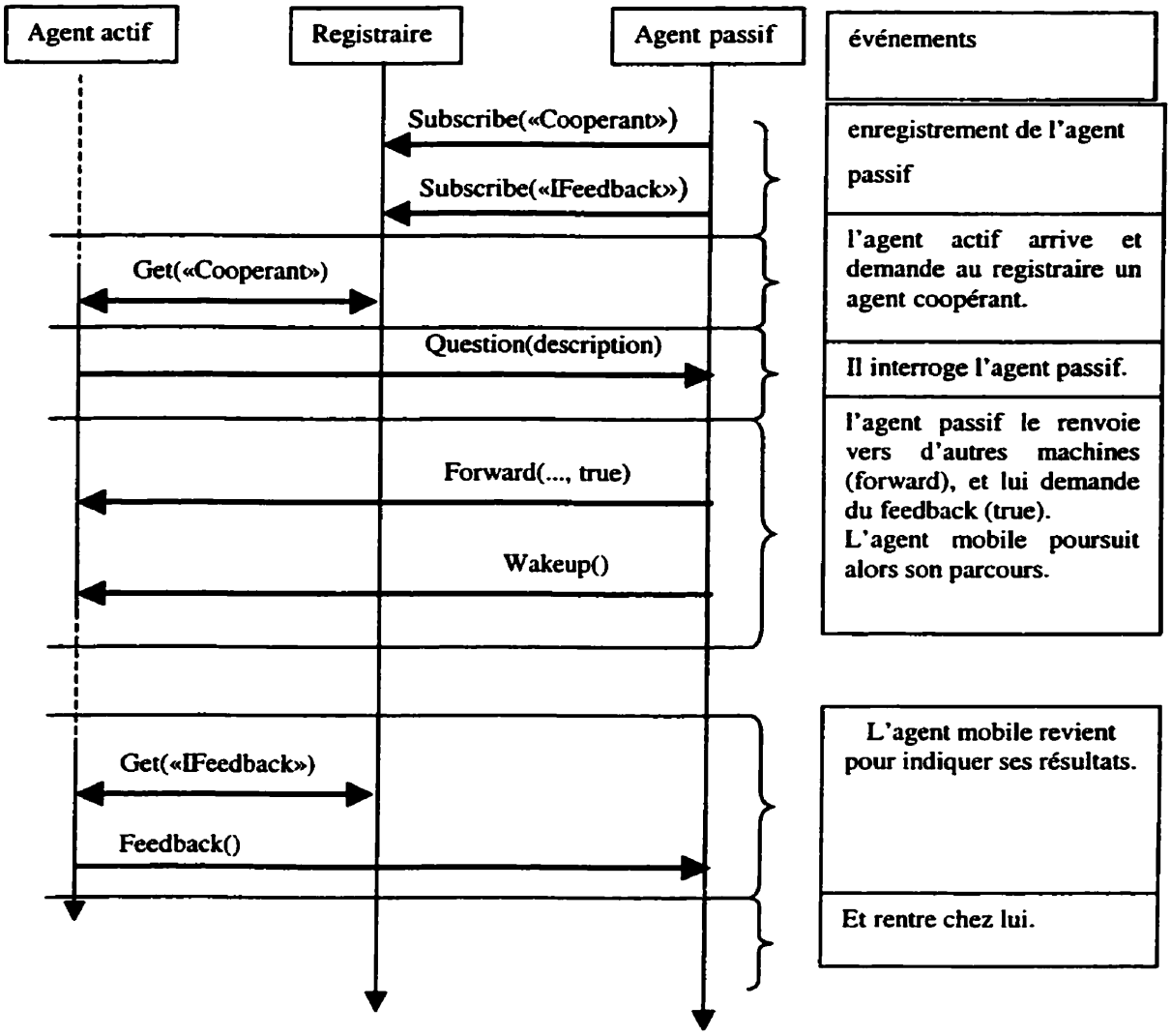


Figure 3.3 Communication entre agents

Le registraire étant en contact permanent avec l'agence, il est également bien placé pour jouer un rôle de «gendarme» en complément des fonctionnalités de sécurité procurées par le système. Il peut surveiller les allées et venues des agents, afin, par exemple, d'éviter les agents indésirables ou les profiteurs qui vont chercher à utiliser les ressources du système sans rien lui apporter, comme on va le voir dans le paragraphe suivant. La Figure 3.4 résume les relations entre les différents éléments du système.

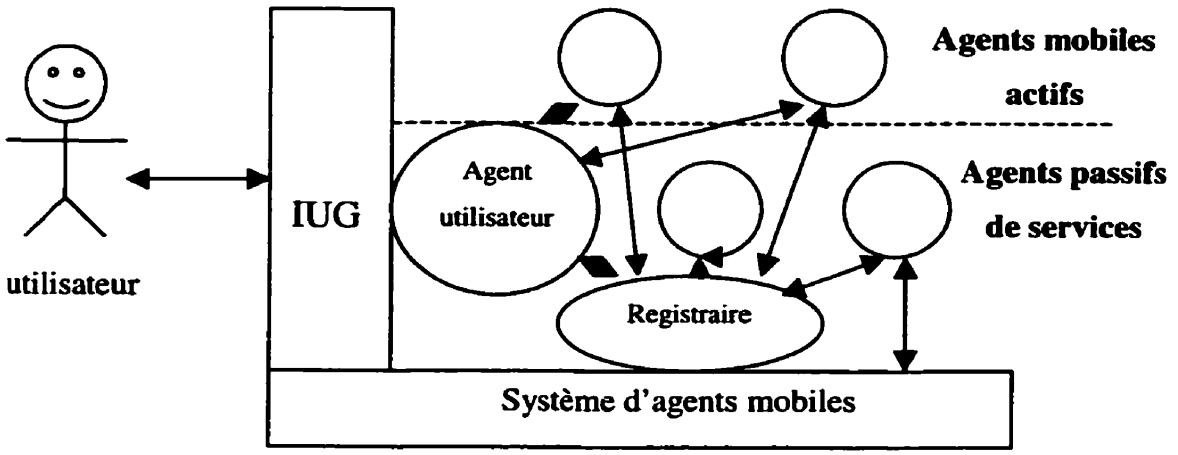


Figure 3.4 Relations entre les éléments de l'architecture

3.1.2 Vers des serveurs actifs

On a vu que le but du concept d'agent mobile est d'amener le calcul aux données plutôt que les données au calcul (Lange, 1998). Ceci est habituellement réalisé en encapsulant tout le code voulu dans un agent mobile qui va voyager sur chaque serveur et accéder aux interfaces qu'ils offrent, idéalement des interfaces de bas niveau, mais souvent des interfaces de haut niveau, destinées à des utilisateurs humains. L'agent doit être rechargé en entier à chaque modification. Il faut noter que Grasshopper garde chaque agent en mémoire cache pour une réutilisation ultérieure, mais qu'il est alors très difficile, voire impossible, de charger une version différente d'un agent qui se trouve déjà en mémoire. L'agent mobile doit souvent surmonter l'inadéquation qui existe entre l'interface offerte par le serveur et ses propres besoins, ce qui entraîne un surcroît de code à transporter à chaque fois. L'architecture proposée ici permet la réutilisation de ce code en l'encapsulant dans un agent séparé qui va être ajouté à l'interface du serveur. Celui-ci va donc gagner dynamiquement de nouveaux services et une nouvelle interface qui s'ajoutent à ce qui existait initialement selon la Figure 3.5. La Figure 3.6 montre comment l'architecture proposée réduit la charge du réseau.

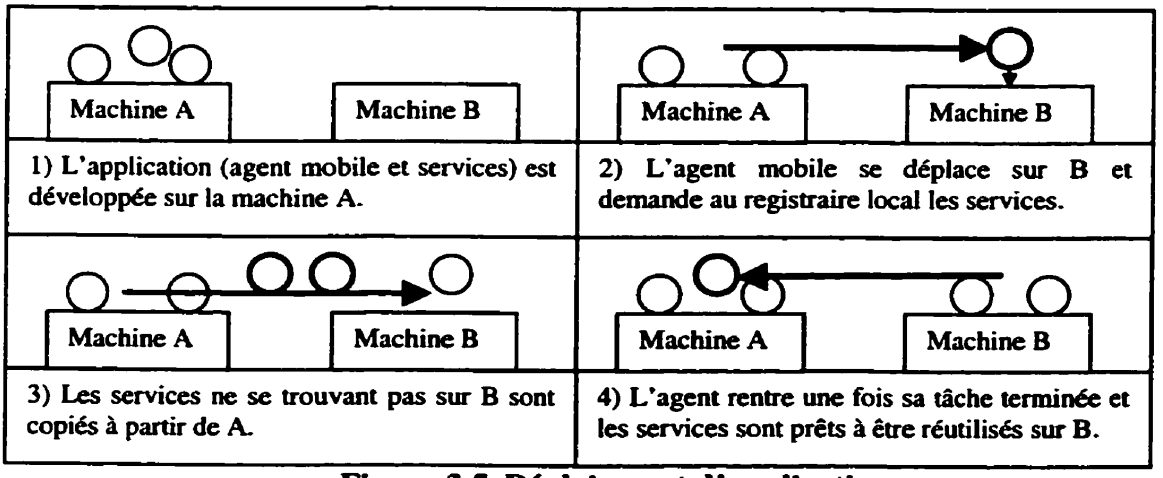


Figure 3.5 Déploiement d'applications

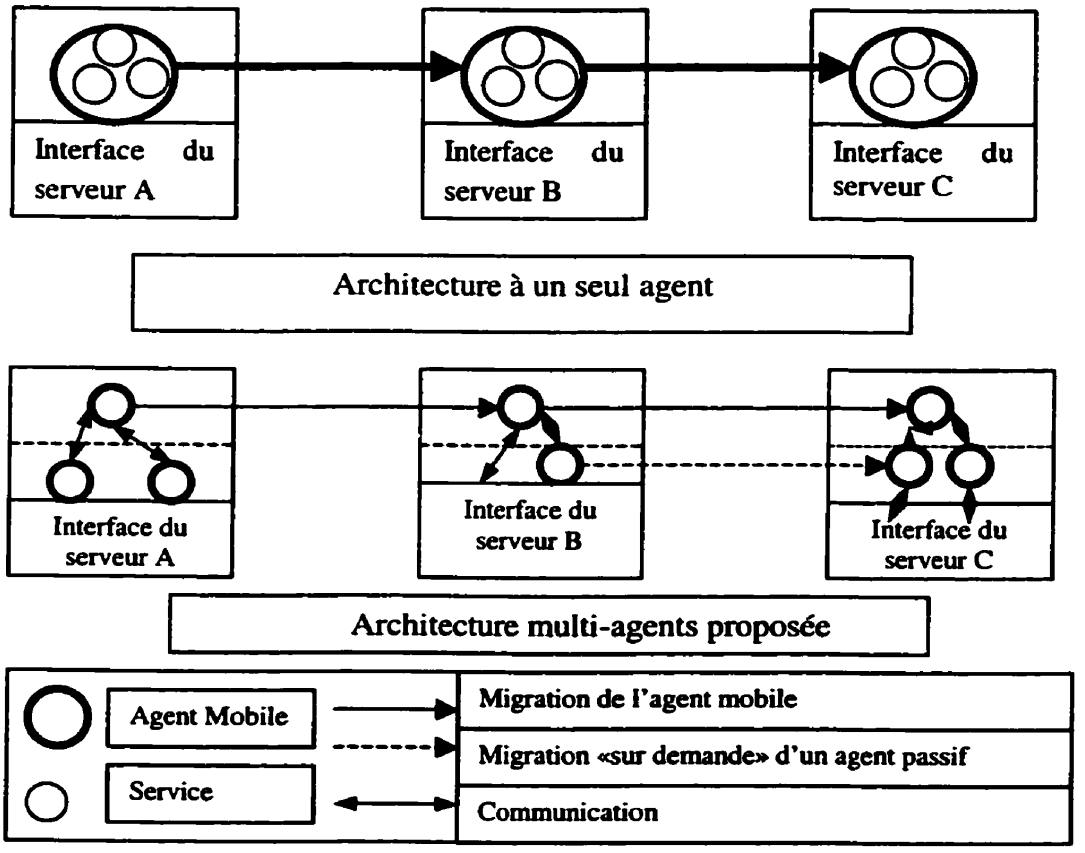


Figure 3.6 Architecture pour la réduction de la charge du réseau

Par la réutilisation du code, on épargne de la bande passante, de même que l'on simplifie la charge de l'administrateur. La mise à jour des services va se faire automatiquement, et celui-ci peut se concentrer sur d'autres problèmes comme le contenu ou la sécurité.

Néanmoins, cela implique que le système est capable de gérer efficacement jusqu'à plusieurs milliers d'agents et de protéger l'hôte des agents malveillants ou gourmands cherchant à profiter des ressources de la machine sans rien apporter. Par exemple, le but d'un serveur d'images ou d'informations est de permettre à un maximum de personnes d'y accéder (au moyen de leurs agents). Il faut donc éviter qu'un seul agent parcoure la base de données du serveur pendant des heures, voire des jours, en utilisant de la mémoire et du temps de calcul au détriment des autres usagers. Il faut aussi supprimer les services qui ne sont pas ou plus utilisés et garder les autres. Pour cela, le registraire est bien placé, comme extension de la plate-forme. Grasshopper (comme la plupart des systèmes d'agents mobiles) permet de connaître tous les agents qui arrivent, sont ou partent d'une agence, qu'ils s'inscrivent ou non auprès du registraire. Le registraire peut conserver ces informations en vue de calculer une fonction de coût qui va représenter le coût d'un agent pour le système. Cette fonction serait de la forme :

$$A/Futil+B*Dutil+C*taille$$

où A, B et C sont des paramètres de normalisation positifs, *Futil*, la fréquence d'utilisation de l'agent, *Dutil*, la durée écoulée depuis sa dernière utilisation, et *taille*, la taille de la mémoire qu'il occupe (code + données). On dira qu'un agent est utilisé lorsqu'il est contacté par un autre agent (par l'intermédiaire du registraire).

Les problèmes de sécurité ne seront pas abordés en détail dans ce mémoire. On a vu que, généralement, le cœur du système est protégé par le mécanisme de bac à sable de l'interpréteur de code mobile (par exemple, la machine Java). Le mécanisme exposé plus haut, ou un mécanisme similaire, peut le protéger d'une utilisation abusive de ses ressources. Enfin, il est possible (c'est inclus dans Grasshopper) de n'accepter que des agents signés par une autorité ou un groupe défini d'utilisateurs en qui l'on a confiance

(Allée, 2001). Toutefois, cette dernière technique peut entraîner un surcoût certain dû à l'utilisation des mécanismes de cryptage.

Il est à noter que cette architecture n'a pas pour but de couvrir la communication et la coordination entre plusieurs agents actifs, mais peut être facilement étendue par des fonctions d'espaces de tuples comme *JavaSpace* ou *Linda* (sites JavaSpace et Linda).

3.1.3 Traitement des connaissances

Nous avons vu de quelle manière l'acquisition et l'utilisation de connaissances peut aider à améliorer les performances d'une application agent mobile. Nous allons proposer dans cette section une façon de coder et de conserver ces connaissances. Nous allons considérer deux sortes de connaissances : la connaissance de la topologie du réseau, et la connaissance de son contenu (agents, places et données).

Le réseau est représenté par un ensemble de places (d'agences) groupées en zones. Ces zones doivent être plus fines et indépendantes des régions de sécurité ou de recherche pour représenter une relation de proximité entre les agences. Globalement, on se ramène à un ensemble d'adresses, agences et zones, réunies par une relation «est dans» qui représente l'appartenance d'une agence à une zone ou l'inclusion d'une zone dans une autre. Une agence peut très bien appartenir à deux zones sans que cela ait d'incidence sur les algorithmes présentés par la suite. On obtient donc un graphe sur lequel on pourra appliquer des techniques heuristiques de recherche plus simples que pour un graphe représentant l'ensemble des liaisons physiques du réseau. De plus, il s'accommode très bien de l'inconnu, les agences dont on ne sait rien peuvent être regroupées dans une zone spéciale «autre» et ce graphe peut être étendu au cours du parcours à l'aide de la relation «est dans». La connaissance complète du réseau n'est absolument pas nécessaire à un agent. La Figure 3.7 illustre la structure d'un tel graphe.

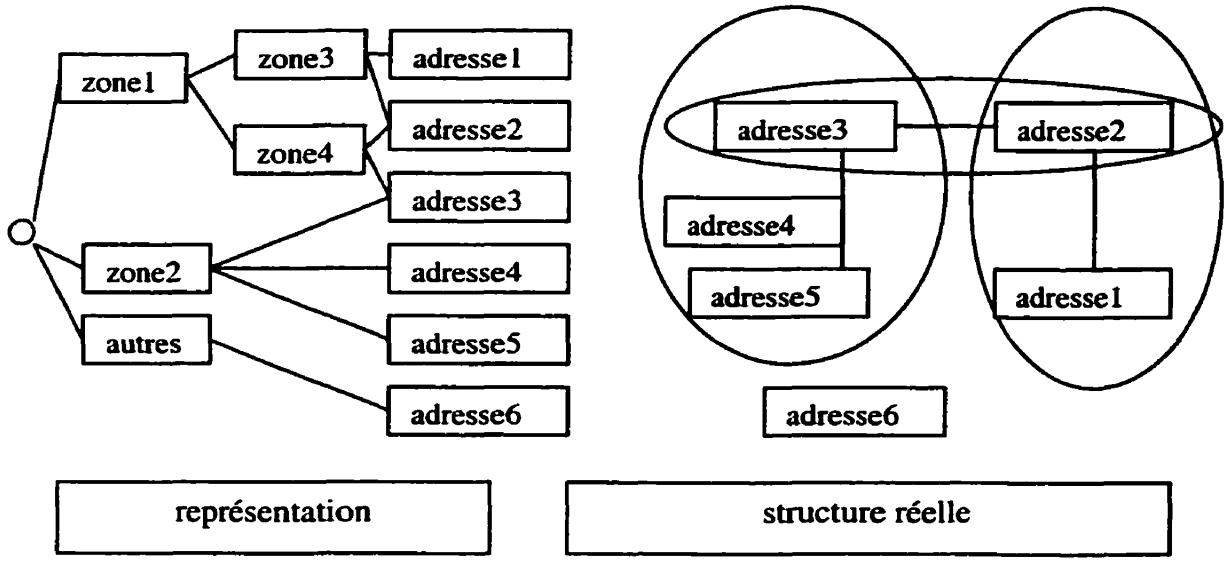


Figure 3.7 Représentation du réseau

Un autre aspect est la représentation du contenu du réseau. Une ressource – place, agent, fichier, base de données - va être représentée par une «adresse». Celle-ci va contenir l'adresse de la machine, ainsi que celle de l'agence où doit se rendre l'agent et le nom de la ressource (avec chemin complet pour un fichier), indiquant également sa nature. Cette adresse devra être accompagnée de renseignements destinés à guider l'agent dans sa recherche d'une ressource utile pour sa tâche. Afin d'alléger l'agent, celui-ci ne doit transporter que le minimum d'informations, les autres étant conservées dans d'autres agents. Ceci a l'avantage de permettre, en outre, à chaque agent de conserver ces informations dans des formats qui peuvent être très différents, mais la communication entre les agents s'en trouve compliquée.

L'utilisation de KQML (Knowledge Query and Manipulating Language) (site KSE) peut fournir un standard appréciable pour la communication entre les agents même s'il nécessite un gestionnaire de communication adapté dans chaque agent. Ce n'est pas à proprement parler un langage de représentation de connaissances comme KIF (développé par la même organisation ARPA), mais un langage de manipulation de

messages, destiné à permettre la communication entre agents. KQML est un langage plutôt qu'un protocole, dans la mesure où il est moins précis et fournit seulement une base de communication. Il n'aborde pas non plus l'aspect de la sémantique et des ontologies. Il s'occupe des problèmes plus concrets suivants : savoir avec qui parler, comment le trouver, comment commencer et prolonger un échange. Les primitives du langage sont appelées "performatives". Concept relié à l'acte de langage, celles-ci définissent les actions que les agents peuvent tenter dans leur communication avec d'autres.

KQML est décomposé en trois couches :

- la couche *contenu* est l'information contenue dans le message codé dans le langage du programme, quel qu'il soit ;
- la couche *communication* code des attributs de bas niveau de la communication, comme les adresses de l'envoyeur et du destinataire, et un identificateur de communication ;
- la couche *message* sert à coder le message en termes d'interactions entre agents et forme le cœur de KQML.

La première fonction de cette couche est d'identifier le protocole à utiliser pour délivrer le message et spécifier une "action de langage", ou "performative", que l'envoyeur attache au contenu, telles que «ask-if», «ask-about», «ask-one», «ask-all», «reply», «sorry», «tell», «achieve», «cancel», «untell», «unachieve», «advertise», «subscribe», «register», «unregister», «forward», «broadcast», «route». Cette couche peut contenir également d'autres caractéristiques du message, comme l'ontologie. L'ensemble de ces caractéristiques permet l'analyse et la délivrance des messages, même lorsque leur contenu n'est pas accessible ou compréhensible. La syntaxe est basée sur une liste entre parenthèses et révèle l'origine Lisp de la première implémentation. Comme elle est très simple, on peut facilement la modifier si nécessaire. La figure 3.8 donne un exemple de message KQML d'un agent *joe* à un agent *stock-server* pour connaître le prix d'une action IBM.

	Couche KQML
(ask-one : sender joe : content (PRICE IBM ?price) : receiver stock-server : reply-with ibm-stock : language LPROLOG : ontology NYSE-TICKS)	communication contenu communication message
reponse à ce message (tell : sender stock-server : content (PRICE IBM 14) : receiver joe : in-reply-to ibm-stock : language LPROLOG : ontology NYSE-TICKS)	communication contenu communication message

Figure 3.8 Exemple de messages KQML

KQML introduit, pour faciliter la communication entre les agents, des *facilitateurs* et des *médiateurs*. Un *facilitateur* est un agent qui s'occupe plus particulièrement d'offrir des services de communication aux autres agents. Dans notre cas, le *Registraire* est un facilitateur.

Toutefois, le principe de notre architecture est que l'agent ne garde que l'essentiel lors de ses déplacements. Le minimum pour un agent mobile est d'avoir un itinéraire avec des priorités accordées à chaque adresse. Ce minimum peut être complété par des informations spécifiques à une application, comme les résultats ou les paramètres d'une recherche. Pour obtenir un itinéraire et attribuer une priorité à chaque adresse, l'agent

doit s'adresser aux agents possédant des connaissances sur le contenu du réseau. Plutôt que de leur demander directement ces connaissances, ce qui suppose l'existence d'un langage de description et d'ontologies communes, l'agent mobile va leur demander un itinéraire qui correspond à son but. L'agent mobile peut exprimer son but dans de nombreux langages de description de connaissances, mais peut aussi se contenter d'exprimer ses demandes sous forme d'une requête textuelle comme celle qu'un humain entrerait dans un moteur de recherche ou celle qu'il a fournie à l'agent. Ceci nous entraîne à considérer des techniques de recherche d'information comme celles présentées à la fin de ce chapitre.

3.2 Application numéro pilote

Le but de cette application est d'aider l'utilisateur à trouver un correspondant pour obtenir des renseignements par téléphone en parcourant une liste de correspondants possibles. Dans cette section, nous en présentons le principe, les choix de conception, ainsi que les modifications apportées à l'application initiale.

3.2.1 Principe

Plutôt que d'avoir à composer une série de numéros jusqu'à trouver le bon, l'utilisateur compose un numéro pilote unique (ou un lien Internet « pilote ») et fournit à l'agent les informations voulues. L'agent va alors chercher lui-même le correspondant susceptible de prendre l'appel et, éventuellement, de fournir les renseignements désirés. C'est une version agent des serveurs vocaux des grandes entreprises et administrations, avec plusieurs améliorations. Notamment, l'utilisateur n'a pas à rester en ligne pendant la recherche. Une fois l'agent envoyé, il peut revenir à son occupation initiale en attendant les résultats de la recherche. Il n'est pas obligé de prendre la communication immédiatement. De plus, ce service offre une interface beaucoup plus personnalisée et conviviale.

L'application se composait, initialement, d'un unique agent mobile transportant les informations de l'utilisateur et une liste de correspondants possibles. L'agent voyage sur

les machines de chaque utilisateur listé jusqu'à trouver la bonne personne. Une des améliorations apportées à l'application initiale est de rendre la liste des correspondants dynamique. Ainsi, l'agent peut être renvoyé vers d'autres correspondants en cours de route, même si ceux-ci n'étaient pas prévus dans la liste initiale, car non connus de l'utilisateur. Cette première modification fait passer les possibilités de l'application dans l'univers des agents mobiles et ouvre la voie à d'autres modifications exposées plus loin.

3.2.2 Choix de conception

Après un premier prototype développé sur Voyager par Bertrand Emako-Lenou (2000), nous sommes passés à Grasshopper qui offrait des possibilités intéressantes pour la conception et le déploiement d'agents mobiles et était également plus fiable.

Un choix important quant à la conception de cette application portait sur l'interface téléphonie - agent. Le domaine considéré est ici celui de la téléphonie sur Internet où l'on fait transiter un appel téléphonique par un réseau IP. Plusieurs protocoles sont à l'essai pour concilier IP et temps réel. Les deux principaux sont H323 et SIP. Pour Glitho (2000), ces protocoles n'atteignent pas les objectifs espérés – supporter une large palette de services et de fournisseurs, création, gestion et personnalisation rapide et simple des services, indépendance au réseau, collaboration avec les services déjà existants. Pour compenser ces faiblesses, Parlay (site Parlay) a été introduit comme une couche supplémentaire au-dessus de ces protocoles pour offrir une interface simple et standard aux fournisseurs de services. C'est une technologie orientée client/serveur. On a donc le choix de relier les agents mobiles uniquement à SIP (l'agent pourrait intégrer un client SIP), à Parlay ou aux deux. La Figure 3.9 résume les différentes possibilités d'implémentation.

Devant la plus grande complexité de SIP et la mauvaise programmation des agents client SIP existants, Parlay s'est imposé. Une interface Parlay générique a déjà été écrite par K. Sylla (Derochers, 2000) en Java et liée au protocole SIP. Elle ne peut cependant pas être mobile, même en partie, Parlay étant destiné à être implanté sur des serveurs, et non des palm-top. L'interface est donc réalisée par un agent statique. Sa propriété

d'agent lui permet d'être lié au système d'agents mobiles, ici Grasshopper, et d'être facilement contacté par d'autres agents, même à distance. Étant sur un serveur Parlay, il peut communiquer facilement avec les protocoles de téléphonie. Cet agent implémente une interface spécifique appelée *IServeur* qui va le caractériser dans l'architecture décrite précédemment.

3.2.3 Modifications apportées à l'application initiale

Dans la première version, l'agent parcourait un ensemble de places déterminé. L'ajout d'une liste dynamique de destinations ouvre la voie à d'autres apports. La section précédente décrit le choix de l'interface avec les fonctions de téléphonie. Nous allons ici plus nous attarder sur le coté recherche d'information. En effet, le but de l'agent est de trouver un correspondant téléphonique, soit un certain type d'information représentée par une adresse IP (dans le cas de téléphonie IP), une réponse booléenne – réponds/réponds pas – et éventuellement d'autres informations extensibles à loisir.

Le répertoire des services de téléphonie actuel, qui nous aide tant bien que mal à trouver ce que nous cherchons, comprend répondeurs, annuaires, serveurs vocaux, messageries, ... Il est normal de songer à adapter ces outils à la téléphonie IP et à notre application en particulier, d'autant que c'est particulièrement simple, l'agent transportant une description du but de l'appel sous une forme textuelle, plus condensée et compréhensible par les machines que la voix. Chacun de ces outils ou services peut être implémenté sous la forme d'un agent qui va communiquer avec l'agent mobile à travers une interface simple décrite avec l'architecture. Le rôle de l'agent mobile va alors se réduire à parcourir une liste dynamique d'adresses, chaque adresse correspondant à un agent humain ou virtuel.

Le but de l'application est lui-même élargi à l'initialisation de tout appel. Elle peut par exemple chercher un utilisateur utilisant plusieurs appareils, chercher une personne occupant une certaine fonction dans une entreprise ou une administration, ou pouvant fournir certaines informations.

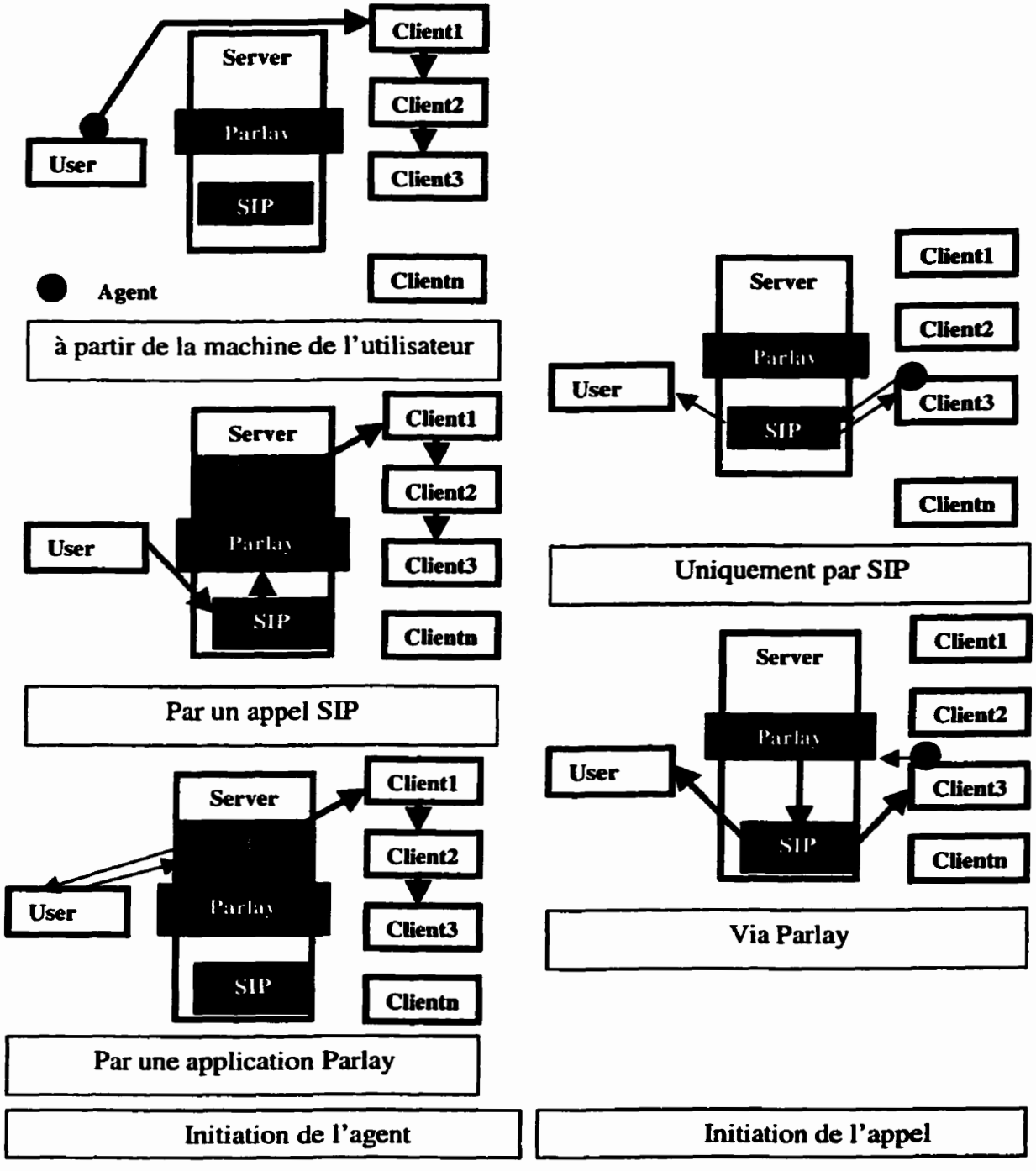


Figure 3.9 Choix de conception de l'interface téléphonie-agents

3.3 Application chercheur d'images sur Internet

Cette application cherche sur un réseau des images correspondant à la requête de l'utilisateur – «clip art», cartes de vœux. Dans la version développée, les images et leurs descriptions sont obtenues via des pages HTML. L'agent voyage sur chaque serveur hébergeant une base d'images et lit les pages HTML offertes par le serveur comme interface avec la base. L'agent retourne ensuite à son point de départ avec les images correspondant à la requête de l'utilisateur. Plus précisément, l'agent parcourt sa liste de destinations, en commençant par une liste initiale de sites connus. Quand il arrive à un site contenant une base d'images, il parcourt les pages HTML qu'on lui a indiquées en cherchant les renseignements selon la méthode décrite plus loin. Il note ensuite les résultats de cette recherche selon les critères fournis par l'utilisateur. Il revient dès qu'il a suffisamment de résultats ou qu'il a fini son parcours.

3.3.1 Interface avec les bases d'images

Une base de données avec une interface JDBC aurait été la bienvenue pour le développement de notre prototype, mais rares sont les sites offrant un accès direct à leur base de données sur Internet, pour des raisons de sécurité. Comme on voulait concevoir une application réaliste, voire immédiatement utilisable, on a choisi comme interface la plus représentée sur Internet : les pages HTML, éventuellement dynamiques. Notre agent devait être capable d'extraire les informations nécessaires – adresses des images et leur description – des pages HTML de la façon la plus simple et la plus générale possible. Pour réaliser cela, on s'est inspirés du langage WONDEL développé à l'Université d'Ottawa par Ouahid et Karmouch (1999). Ce langage sert à décrire le contenu de fichiers HTML de façon à en extraire l'information souhaitée.

Dans la version développée à Ottawa, WONDEL sert à stocker la méta-information (l'information sur l'information) dans des fichiers XML qui forment une structure d'arbre. Les fichiers « feuille » relient les informations aux documents en indiquant où elles se trouvent dans le document en terme de «structure» (ce sera précisé plus loin). Les fichiers « nœud » relient ces informations entre elles. Ce langage a été

développé pour être utilisé par un agent parcourant un site important en vue d'indexer l'information qui s'y trouve dans une base de données. Il utilise la similitude de structure qui existe entre différents fichiers d'un même site pour réduire considérablement le volume d'information initiale (où trouver l'information) nécessaire à l'agent. En effet, ces fichiers étant souvent écrits par un petit nombre de personnes et de logiciels, ils tendent à être construits selon une structure commune. Par exemple, pour une liste des employés d'un département, on peut avoir une page telle celle schématisée à la Figure 3.10.

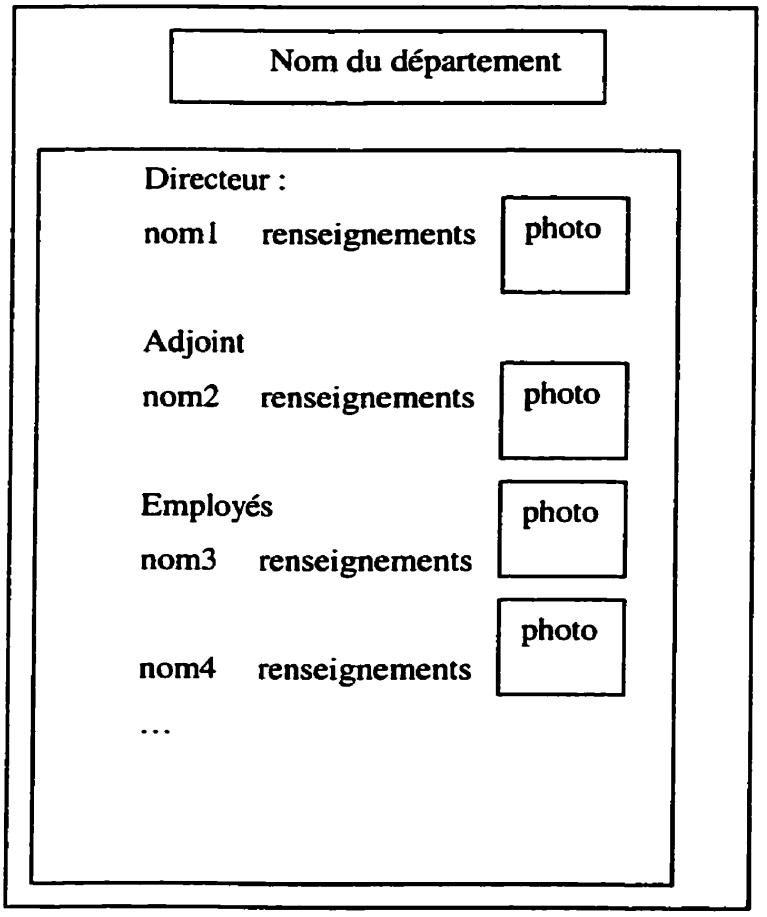


Figure 3.10 Exemple de page HTML

Si le site n'est pas trop mal fait, cette structure peut se retrouver au niveau du code HTML, comme indiqué à la Figure 3.11.

```
<HTML>
...
<body>
  <h1>nom du département</h1>
...
  <h2>fonction</h2>
  <li>nom1 renseignements<img source=[adresse de la photo]></li>
...
```

Figure 3.11 Exemple de code HTML

On pourrait décrire le parcours de cette structure par un agent en pseudo-code par la Figure 3.12.

```
Chercher <h1> et créer les résultats
Contenu de <h1>=nom du département
Pour chaque <h2>
  Contenu de <h2>=fonction
  Pour chaque <li>
        Contenu=nom renseignements
        Trouver un tag <img>
        Valeur de l'attribut src=photo
        Enregistrement des valeurs courantes
```

Figure 3.12 Algorithme de parcours d'une page HTML

Ces informations pourraient être codées tel quel dans une structure XML, mais la présence de boucles imbriquées complique l'algorithme de parcours et risque d'introduire des redondances. La simplification adoptée par Ouahid et Karmouch (1999) est de ne faire que la boucle la plus « petite », ici sur les tags <li>. Les autres renseignements seront recherchés à partir de là dans la structure d'arbre du document HTML. Leur fichier WONDEL suit la structure de l'enregistrement final (un tuple composé de plusieurs champs). Cette structure est décrite plus en détail dans (Ouahid et Karmouch, 1999).

Dans le but de simplifier les algorithmes et d'accélérer le parcours des pages, au risque de perdre certains renseignements de moindre utilité, nous avons adopté une structure plus proche du pseudo-code en « reproduisant » dans le fichier de description la structure qui contient les renseignements dans la page et où se trouvent les renseignements voulus au sein de cette structure. Pour le fichier de l'exemple, on aurait le fichier de la Figure 3.13.

```
<h2>
<li>
<texte nom>
</texte>
<img>
<attributs>
<src photo>
</src>
</attributs>
</img>
</li>
</h2>
```

Figure 3.13 Fichier de méta-information

Dans cette figure, h2 indique le départ de la boucle : pour chaque <li>, l'agent enregistre les valeurs trouvées à chaque fois qu'il arrive à parcourir toute la structure. On voit qu'on perd une partie de l'information, mais qui n'est pas importante pour la recherche. Ce fichier pourra alors être utilisé pour toutes les pages construites sur le même modèle. Ceci est particulièrement intéressant pour fouiller des sites contenant des pages générées dynamiquement à partir de bases de données. Il suffit d'écrire un tel fichier pour chercher l'information dans toutes les pages du site ayant la même structure (une infinité pour des pages générées dynamiquement).

3.4 Algorithmes de recherche d'information utilisés

Les deux applications considérées dans la section précédente étant des applications de recherche d'information (RI), elles gagneraient à être dotées d'une plus grande intelligence et de capacités d'apprentissage. Malheureusement, ajouter ces algorithmes directement dans les agents augmenterait considérablement leur taille et réduirait leurs performances de façon prohibitive. Pour remédier à cela, nous proposons d'appliquer aux agents mobiles des algorithmes d'apprentissage par « feedback » en espérant des pertes de performances minimes.

Nous essaierons d'appliquer et d'évaluer cette technique de feedback sur la requête, mais surtout sur la description des documents eux-mêmes. Cela est rendu possible et même nécessaire par le fait que, dans le cas de notre application, l'indexation des documents n'est pas réalisée au niveau du serveur, mais est créée par l'application elle-même en dehors des bases de données. Si une indexation existe au niveau du serveur, elle peut toutefois servir de point de départ appréciable.

Dans l'architecture multi-agent développée, les connaissances et les algorithmes de RI sont regroupés dans une classe d'agent : le «KnowAgent», qui va interagir avec l'agent mobile. Les algorithmes utilisés vont refléter les caractéristiques dynamiques et changeantes de l'environnement où l'application est censée être déployée.

3.4.1 Choix généraux

Nous avons choisi le modèle espace-vectoriel comme modèle de représentation de l'information pour sa plus grande souplesse d'utilisation. Car, il est plus adapté à des requêtes en langage naturel, et surtout pour l'apprentissage. En effet, le mécanisme de feedback permet l'apprentissage, par un renforcement progressif des mots réellement significatifs de chaque document, de façon simple et éprouvée (Nilsson, 1965).

3.4.2 Adaptations

Une différence importante par rapport à une situation classique de RI est que l'on ne connaît pas les documents a priori et qu'on ne peut pas non plus les indexer par les moyens classiques (images ou personnes). On ne se trouve pas non plus dans une situation de filtrage d'information car l'information ne vient pas «toute seule» à nous. Il faut un moyen d'y accéder. Il faut donc réaliser un index, malgré les problèmes que nous venons de voir. Cet index va être réalisé en partie au niveau des serveurs par les annuaires en ligne ou un mécanisme de recherche classique, et au niveau des clients par le *KnowAgent*, par feedback et apprentissage. Ce sont ces derniers qui vont nous intéresser maintenant.

Cet agent n'a comme point de départ que quelques adresses indiquées par l'usager. Il ne va pouvoir apprendre que par feedback, n'ayant aucune autre information sur le reste du réseau. Il va enregistrer l'itinéraire final de l'agent mobile pour ajouter les adresses où l'agent a été renvoyé, avec des poids initiaux nuls. On va renforcer les mots de la requête dans la représentation d'une adresse réponse et les rabaisser pour une adresse où l'agent n'a pas eu de réponse.

Avec les notations de la section 2.4.3 (Q est le vecteur requête, D_i le vecteur document à la i-ème itération, T le seuil de décision et A l'ensemble des bonnes réponses) :

$D_i = D_{i-1} + cQ$ si $M(D_{i-1}, Q) - T \leq 0$ si $D \in A$

$D_i = D_{i-1} - cQ$ si $M(D_{i-1}, Q) - T > 0$ si $D \in A'$

On va ainsi réaliser, par renforcements successifs, un index de départ correspondant aux besoins de l'usager.

L'indexation est habituellement réalisée en plusieurs étapes. La plus importante est certainement la mise en paramètres. Dans le modèle espace-vectoriel, il s'agit de calculer le poids associé à chaque mot. La formule couramment adoptée est celle du *tfidf* (term frequency inverse document frequency), à savoir la fréquence du terme dans le document divisée par la fréquence du terme dans l'ensemble des documents. On a ainsi une représentation de l'importance du mot dans le document en réduisant l'importance des termes les plus courants. Cela suppose de connaître l'ensemble des documents. Or, nous n'avons accès qu'à *tf*, qui sera donc retenu pour l'indexation des documents textuels (dont les requêtes).

Beaucoup utilisent aussi les techniques de «stop list» et de «lemmatisation» (Altavista, Yahoo). La première consiste à retirer au début du mécanisme d'indexation les mots trop courants tels que les articles. La deuxième consiste à ramener toutes les formes grammaticales d'un même mot à un radical (lemme). Pour des raisons de simplification, nous n'emploierons aucune de ces deux techniques. Certains moteurs de recherche (AliWeb, HotBot, OpenText) choisissent d'ailleurs de garder tous les mots. Cette élimination peut aussi être faite après indexation (chose bien utile dans notre cas : l'indexation se fait au fur et à mesure) en tronquant le dictionnaire utilisé pour ne garder que les mots les plus significatifs, i. e. ceux qui distinguent les documents les uns par rapport aux autres. Comme l'indexation est réalisée par feedback dans notre cas, ceci devrait permettre de ne retenir que les mots réellement significatifs : ceux que l'utilisateur utilise effectivement.

Chapitre IV
Implémentation et résultats

Afin de valider les concepts présentés dans le chapitre 3, nous avons implémenté les deux applications qui y sont décrites en utilisant l'architecture présentée, puis nous avons réalisé des mesures de performance. Un des premiers choix d'implémentation est celui de la plate-forme. Après le développement d'applications simplifiées sous Voyager, nous avons travaillé sous Grasshopper. Les mesures ont été finalement réalisées sous Grasshopper. Comme l'intérêt de l'architecture présentée en terme de réduction de la charge réseau découle directement de sa conception, nos mesures vont s'orienter vers l'évaluation des différents algorithmes de routage et de recherche d'information décrits dans le chapitre 3. L'agent actif peut être considéré dans les deux cas comme un agent de recherche d'information. Nous allons comparer les performances de trois versions de cet agent. La première parcourt son itinéraire dans l'ordre d'arrivée. La seconde regroupe ses arrêts par régions géographiques. La troisième va, en plus, noter chacune des régions par ordre d'intérêt afin de minimiser ses déplacements inutiles. Chacune des fonctionnalités ajoutées augmente la taille de l'agent mais laisse espérer une meilleure optimisation des déplacements. Dans ce chapitre, nous traiterons des choix d'implémentation faits, des mesures réalisées et de leur interprétation.

4.1 Choix d'implémentation

L'architecture présentée au chapitre 3 a été implémentée en grande partie. En particulier, la communication KQML entre les agents n'a pas été implémentée, mais la structure choisie, par interfaces, peut être facilement adaptée aux messages KQML. Chaque fonction de l'interface serait alors représentée par une commande textuelle suivie des arguments appropriés. Dans ce qui suit, nous présentons la liste et la structure des agents composant le sous-ensemble de l'architecture que nous avons implémenté. De nouveaux agents peuvent y être ajoutés sans modification majeure.

4.1.1 Classes génériques

Nous regroupons sous le terme de «classe générique» les classes qui servent à chacune des applications et qui par conséquent ne sont spécifiques à aucun agent ou choix d'algorithme autres que ceux retenus lors de la conception de l'architecture. Ce sont essentiellement : la classe *Address* qui peut repérer une ressource réseau, et la classe *Lien* qui peut ajouter à cette adresse une priorité et des informations sur sa position dans le parcours de l'agent. La Figure 4.1 décrit ces deux classes. À celles-ci s'ajoute la classe *Job* qui permet de représenter une «tâche» d'un agent par une chaîne de caractères décrivant la tâche et l'identifiant d'un agent. Elle est particulièrement utile pour les agents passifs qui peuvent recevoir des requêtes de plusieurs agents simultanément, mais elle peut être également utilisée par les agents actifs, notamment dans un but d'ordonnancement des tâches. Cette classe peut être considérée comme une représentation interne simple d'une communication par messages KQML, car elle contiendrait alors l'expéditeur du message et le message lui-même.

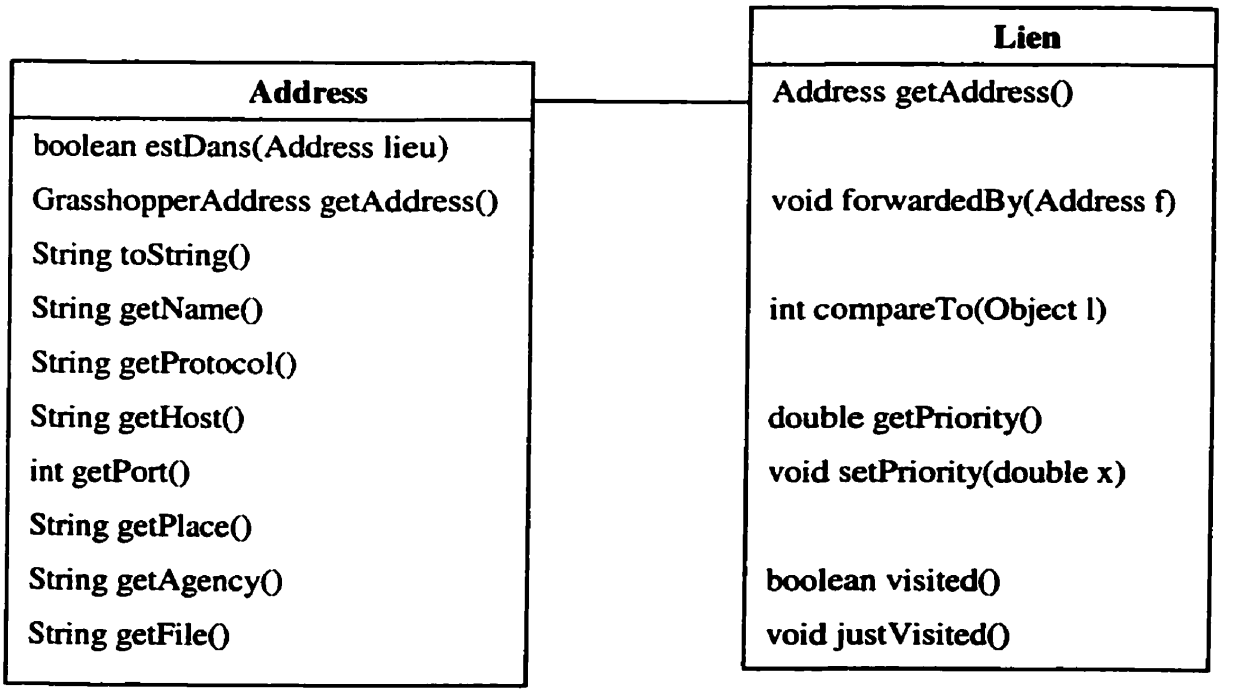


Figure 4.1 Interfaces des classes *Address* et *Lien*

Une caractéristique de la classe *Address* est de pouvoir être construite à partir d'un grand éventail d'objets Java représentant autant de concepts, d'où un grand nombre de constructeurs (non représentés sur la figure). On retrouve également dans la description des classes Grasshopper, «GrasshopperAddress» qui note une adresse sous Grasshopper dans le format hôte/agence/place, ou «Identifier» qui sert à identifier un objet de façon unique dans le système Grasshopper.

4.1.2 Interfaces

Une particularité de l'architecture que nous avons proposée est que l'on ne va pas chercher un agent à partir d'une description ou d'un nom, mais à partir de l'interface qu'il présente. Ceci est motivé par le fait qu'il faut connaître l'interface d'un agent pour pouvoir créer un «proxy» de communication vers celui-ci dans Grasshopper. Une autre raison, plus conceptuelle, est qu'un agent actif arrivant à une nouvelle agence ne va pas y chercher un agent particulier, sauf exception, mais un agent capable d'effectuer une certaine tâche correspondant à une interface particulière. Les interfaces les plus importantes sont : *IRegistraire*, *Cooperant* et *IAgentChercheur*.

IRegistraire est l'interface offerte par les Registraires. Elle doit permettre l'ajout et le retrait de services, ainsi que la mise en communication d'un agent mobile avec un agent procurant le service qu'il recherche. Elle comprend les fonctions suivantes :

- *void subscribe(Identifier agent, String style)* qui ajoute un service "style" procuré par l'agent "agent". L'opération inverse est réalisée par la fonction *void unsubscribe(Identifier agent, String style).*
- *boolean question(Identifier chercheur, String destinataire, String description)* est la fonction de messagerie par excellence, qui envoie un message «description» d'un agent «chercheur» à un service «destinataire». On parle ici d'un service et non d'un agent en particulier. C'est au registraire de trouver l'agent procurant ce service.

- *Identifier get(String destinataire)* permet à un agent d'avoir l'identifiant d'un agent procurant un service «destinataire» sans avoir à envoyer explicitement de messages.
- *boolean get(String destinataire, GrasshopperAddress lieu)* : ici on demande le service sur une autre machine «lieu». L'agent procurant le service (s'il existe) va donc se déplacer ou se copier sur la machine lieu, ou encore procurer un «proxy» de communication et rendre le service à distance s'il ne peut pas se déplacer ou que la politique du réseau l'interdit.

L'interface générale des agents passifs est *Cooperant*. Elle doit permettre la communication ainsi que la recherche et la manipulation de l'agent par le registraire. Cette interface intègre les fonctions suivantes :

- *void question(Identifier chercheur, String description)* est la fonction de recherche et de communication. Les résultats et messages ultérieurs sont ensuite communiqués directement à l'agent «chercheur» qui les a demandés.
- *boolean go(GrasshopperAddress lieu)* est la fonction appelée pour demander à un agent de se déplacer sur une autre machine (ici «lieu»), pour y apporter un service ou dans le cas d'un arrêt de la machine sur laquelle il réside.
- *AgentInfo getInfo()* : donne les informations Grasshopper de l'agent, en particulier son identifiant qui permettra de le joindre par la suite.

De nombreuses autres interfaces peuvent étendre *Cooperant*, comme *IUserGUI*, l'interface de l'interface graphique de l'utilisateur, qui présente peu d'intérêt théorique. C'est l'interface qui permet la communication entre les fenêtres graphiques et l'agent utilisateur.

IAgentChercheur est l'interface du *HuntGroup* (l'agent mobile). Elle est ici plus adaptée aux agents de recherche d'information et surtout à l'agent mobile de l'application HuntGroup développée. Elle peut être facilement modifiée ou héritée pour d'autres applications. Elle contient les fonctions suivantes :

- *void forward(Address [] stops, boolean notify)* et *void forward(Lien[] stops, boolean notify)* servent à renvoyer l'agent vers de nouvelles adresses. Le

paramètre booléen "notify" sert à indiquer à l'agent mobile que l'on désire être informé des résultats de sa recherche en vue de compléter et mettre à jour ses informations. En pratique, l'agent revient, une fois sa tâche accomplie, montrer son parcours.

- *void wakeUp()* réveille l'agent qui attend un événement ou un résultat.
- *String getCallDescription()* donne la description de la recherche donnée à l'agent.
- *void setResponse(boolean x, boolean notify)* sert à donner une réponse à l'agent. Le paramètre "notify" a le même sens que dans les fonctions *forward*.

De nombreuses autres interfaces peuvent être ajoutées pour chaque nouveau service. Il suffit pour cela de rendre disponible une implémentation de ce service qui sera ensuite copiée sur les autres serveurs sur demande (voir 3.1.1 pour l'algorithme de recherche de services). Un problème se pose lorsque deux applications veulent utiliser le même nom d'interface pour deux services différents (ou deux versions différentes d'un même service). À ce moment, l'application arrivant en deuxième sur un serveur se verra proposer un agent implémentant la première interface, alors qu'elle espérait la première. Grasshopper risque de générer une erreur "d'internalisation" pendant le déplacement même de l'agent, en voyant deux versions différentes d'une même interface/classe. Ce problème ne peut être résolu pour l'instant que par une gestion des interfaces des services au niveau du réseau entier.

4.1.3 Agents

Le rôle principal du registraire est d'enregistrer les agents offrant un service spécifique sur la même agence et d'offrir une fonction de recherche de ces agents. Si l'agent cherché ne se trouve pas sur la machine, le registraire peut le chercher et le copier sur des machines «voisines» connues, ce qui nécessite une certaine connaissance géographique du réseau; il peut être également configuré différemment selon la politique du réseau local. Le registraire offre l'interface *IRegistraire*. C'est un agent stationnaire,

même s'il pourrait être copié sur une machine similaire à la première. La Figure 4.2 montre la structure de l'implémentation du registraire.

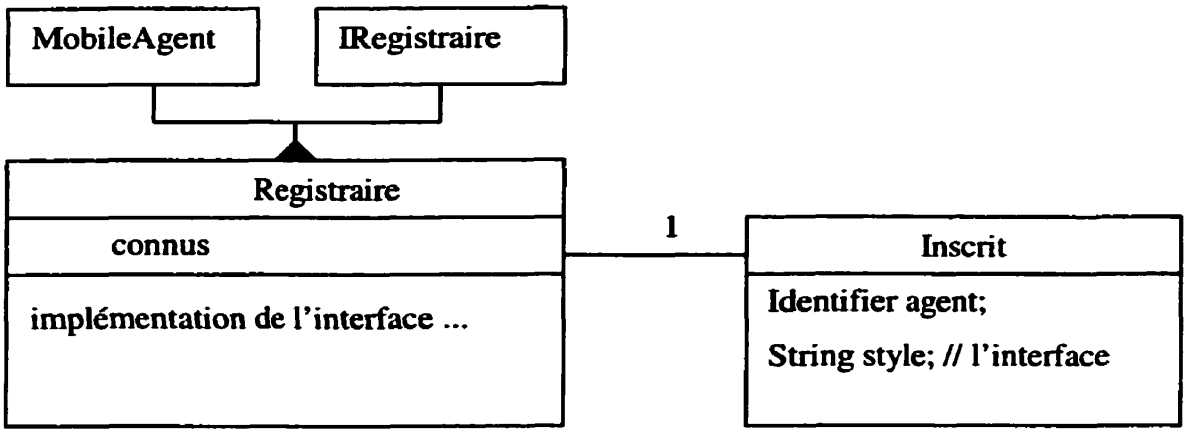


Figure 4.2 Structure du registraire

Le «GUIAgent» fait interface avec l'utilisateur. Il sert idéalement à fournir une interface unique et personnalisée à plusieurs applications. Il implémente l'interface «IUserGUI» et est stationnaire du fait de son caractère graphique et personnel. Il faut noter que les classes graphiques (Frame, ...) sont souvent difficiles à transporter, sans compter les limitations dues aux machines mêmes («Personal Java», destiné aux petites machines comme des palmtops, ne supporte pas la librairie «swing»).

Le *HuntGroup* est l'agent mobile et actif qui est au centre des applications développées. Il implémente l'interface *IAgentChercheur*. Il transporte l'itinéraire et les résultats (comprenant les solutions et sa «connaissance» du réseau). Il assure les fonctions de mobilité dans les applications développées, laissant les fonctions plus spécialisées aux agents passifs et ne transportant que le minimum nécessaire à l'exécution de l'application. La Figure 4.3 montre la structure de l'agent *HuntGroup*.

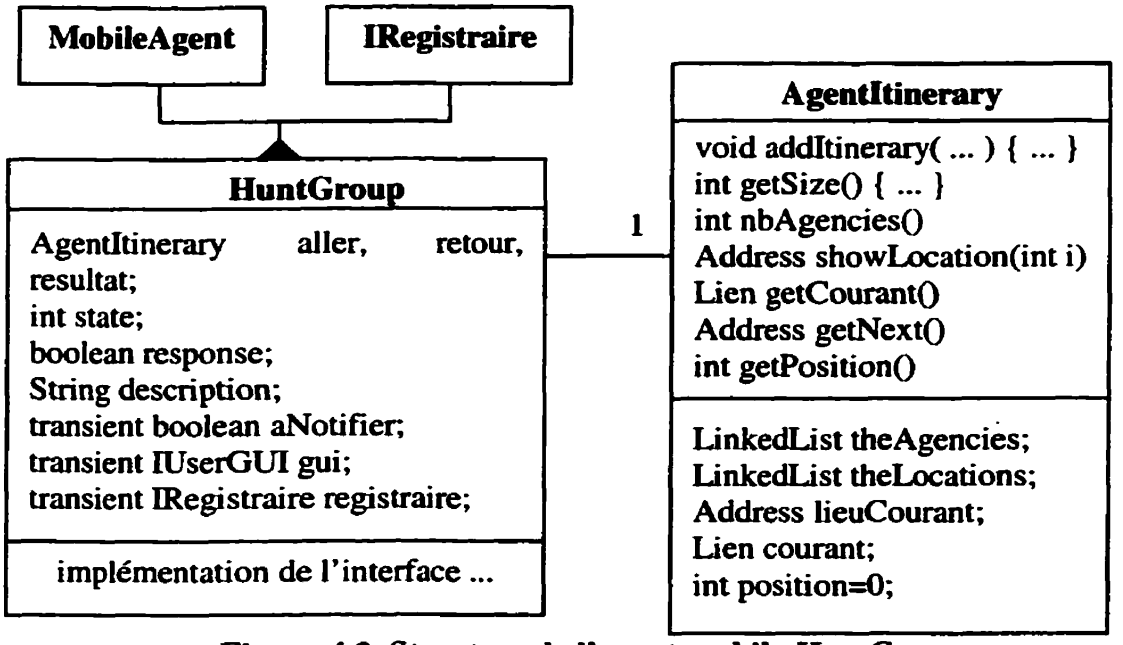


Figure 4.3 Structure de l'agent mobile *HuntGroup*

On peut remarquer que certaines variables du *HuntGroup* sont marquées «transient» car elles ne sont pas transportées. Elles servent en effet à accéder au GUI et au registraire local. Ceux-ci changeant à chaque déplacement, il est inutile, voire dangereux de les transporter. On a également choisi de doter l'agent de plusieurs itinéraires. Ceci est dû au fait que, dans *AgentItinerary*, les destinations sont classées non par ordre «chronologique» mais par ordre de priorité en fonction d'une certaine tâche. Si l'agent doit accomplir successivement plusieurs tâches, il lui faut donc un sous-itinéraire pour chacune de ces tâches, contenant les destinations qui l'intéresseraient pour cette tâche là. Il va donc conserver plusieurs itinéraires qu'il va parcourir successivement. On retrouve dans *AgentItinerary* des fonctions de parcours d'itinéraire dont le corps va varier en fonction de la version développée. Les deux listes *theLocations* et *theAgencies* conservent respectivement les adresses des destinations de l'agent et des zones de proximité géographique que l'on a définies à la section 3.1.3.

Le dernier agent important de notre architecture, *KnowAgent*, conserve les connaissances. Il est semi-stationnaire (il se déplace uniquement sur commande, et de

préférence par copie) et implémente l'interface *Cooperant*. C'est donc un agent passif. Son rôle est de conserver des connaissances et de procurer aux agents mobiles celles dont ils ont besoin. À cet effet, nous allons utiliser les techniques de recherche d'informations décrites précédemment. Cette tâche va être dévolue à la classe *KnowManager*, la classe *KnowAgent* s'occupant de la gestion des communications et des tâches. Nous allons utiliser une classe *Vecteur* qui va représenter un vecteur de nombre (doubles) pouvant être grand mais contenant beaucoup de valeurs nulles, comme c'est le cas en recherche d'information. Ces valeurs vont être conservées dans une table de hachage (HashTable). La Figure 4.4 représente l'organisation de cet agent.

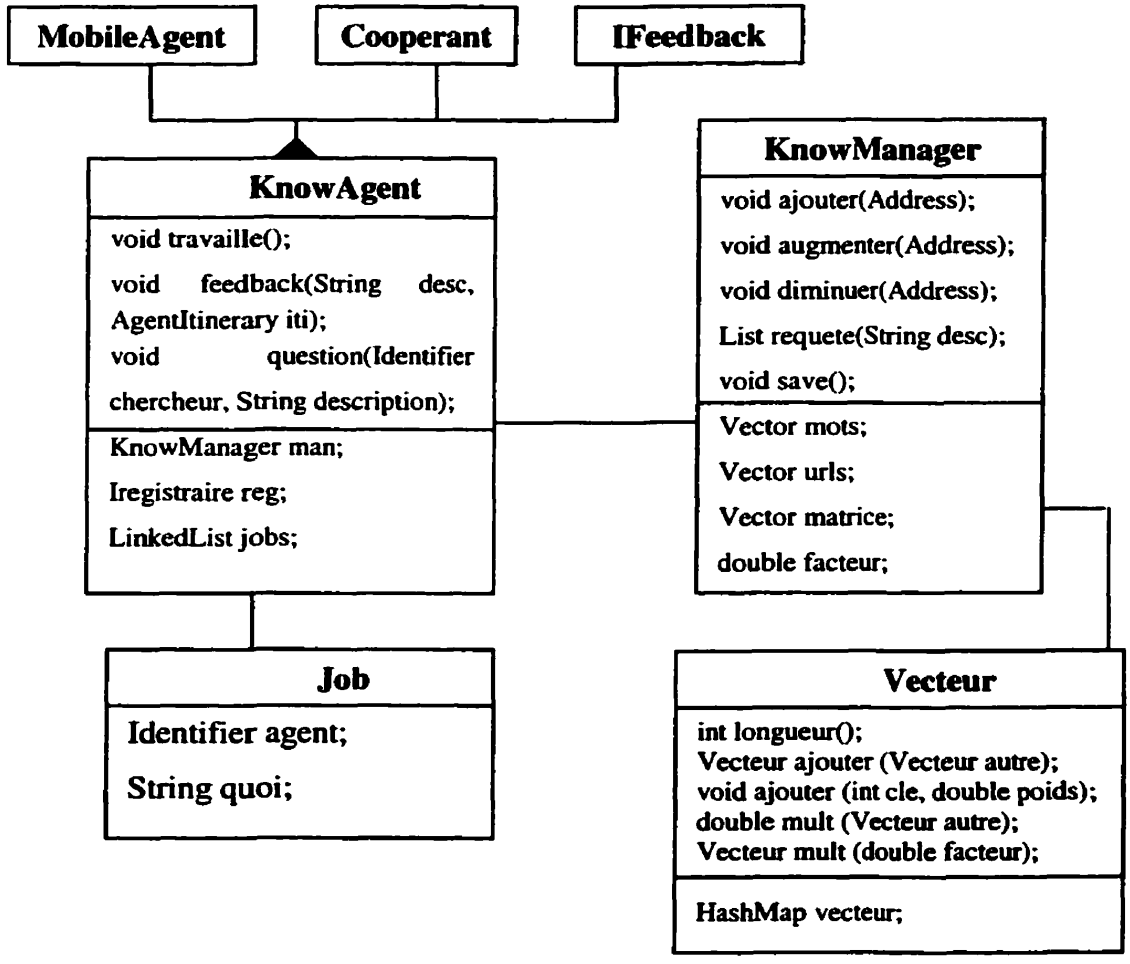


Figure 4.4 Structure du *KnowAgent*

De nombreux agents peuvent être ajoutés pour assurer les services nécessaires aux applications. Ils vont être généralement des agents passifs semi-stationnaires, implémentant l'interface *Cooperant* ou une interface héritée de celle-ci. Nous avons implémenté par exemple un agent *Répondeur*, qui doit répondre aux agents à la place d'un utilisateur humain, et un agent *Chercheur* dont le but est de chercher des informations dans des fichiers HTML (voir 3.3.1).

4.1.4 Environnement d'implémentation et de test

Après le développement de versions simplifiées des applications sous Voyager, d'ObjectSpace, nous avons utilisé Grasshopper comme plate-forme d'agents mobiles pour faire l'implémentation de notre protocole d'enregistrement d'itinéraire. Nous avons opté pour cette plate-forme car elle est utilisée par la firme Ericsson qui est associée à ce travail de recherche. Elle a été développée par la société allemande IKV, la première version a été disponible en août 1998. Il est à noter que l'utilisation de Grasshopper est gratuite pour des fins de recherche. Le langage de développement de la plate-forme est le Java, particulièrement en raison de sa portabilité. Grasshopper est conforme au standard de l'OMG (Object Management Group) sur les agents mobiles, i.e. le standard MASIF (Mobile Agent System Interoperability Facility). Ce dernier a été conçu pour assurer l'interopérabilité entre les différentes plates-formes d'agents mobiles. Grasshopper est un environnement d'agents répartis (Distributed Agent Environment ou DAE). Il est composé de régions, de places, d'agences et de deux types d'agents, stationnaires et mobiles.

Pour mesurer la taille des données transportées sur le réseau, nous avons utilisé 3 machines Windows NT 4.0 Workstation munies d'un processeur Intel Pentium II 400. Le réseau utilisé est un réseau local de type Ethernet 100 Mbps. Les mesures de longueur de trajets ont été réalisées à l'aide de classes Java simulant le déplacement de l'agent dans un réseau, ceci afin d'automatiser les mesures sans avoir à utiliser un environnement de test réparti.

4.2 Évaluation de performance

Comme indiqué précédemment dans ce mémoire, les mesures de performance vont porter essentiellement sur la charge du réseau, car c'est la variable que l'on cherche à optimiser par cette architecture. On peut noter que le temps d'exécution reste inférieur à la seconde, ce qui est tout à fait acceptable pour des applications dans lesquelles le temps de réponse des humains reste le facteur limitant. Nous allons d'abord voir les variations induites par les différents choix d'implémentation, puis par les algorithmes de recherche d'information. Nous allons ensuite réunir les deux à travers différents scénarios.

4.2.1 Mesures de transport

Cette première série de mesures vise à comparer différentes versions de l'application «HuntGroup» en terme de taille de code déplacé par trajet. Tout d'abord, il s'agit d'isoler les paramètres propres au système d'agents mobiles utilisé (ici, Grasshopper). Lorsque l'on utilise des régions Grasshopper pour repérer les agences et les agents ou appliquer une politique de sécurité commune, les agences doivent s'inscrire auprès de la région lors de leur connexion au réseau, ce qui peut être assez fréquent dans le cas de machines mobiles (téléphones sans fil, palmtops, ...). Cependant, elles ne sont pas nécessaires au fonctionnement du système. On a ainsi mesuré le coût de l'inscription d'une agence à la région, soit 14 Ko dans le sens *agence vers région* et 24 Ko dans l'autre sens : 38 Ko au total. Nous verrons qu'en comparaison des coûts de déplacement d'un agent, c'est une valeur assez grande, modérée toutefois par la moins grande «mobilité» des agences (fréquence d'ajout/suppression d'agences).

Une autre caractéristique de Grasshopper est le cache. Les classes et les agents chargés dans une agence sont gardés en mémoire cache pour une réutilisation ultérieure, ce qui pose des problèmes de mise à jour mais permet d'économiser de la bande passante en ne chargeant que les données de l'agent. On va le constater dans la différence entre les mesures du premier chargement et les suivantes.

Nous allons maintenant considérer différentes versions de l'application «HuntGroup». La version de base ne comporte qu'un seul agent comprenant et transportant le GUI et toutes les adresses des correspondants possibles. La taille de l'ensemble des classes est 16.4 Ko. Le Tableau 4.1 résume ces mesures.

Tableau 4.1 Mesures de l'effet du cache et de la région Grasshopper

	Sens	Aller (Koctets)	Aller - Retour
1er envoi avec la région	1->2	26.5	36
	2->1	14.4	26
	Total	40.9	62
2eme envoi	1->2	8.4	17.5
	2->1	7.4	18
	Total	15.8	35.5
sans la région : (1er envoi)	1->2	-	24
	2->1	-	12.8
	Total		36.8

Ces mesures montrent un gain moyen d'un facteur 2 du au cache de Grasshopper, mais aussi lors de la non utilisation de régions. Il est donc intéressant de ne pas utiliser les régions, d'autant que dans l'architecture développée, les registraires prennent les mêmes fonctions (recherche d'agents, sécurité). Les mesures suivantes ont été faites sans utiliser les régions Grasshopper.

Une version améliorée de cet agent consiste à séparer l'interface graphique de l'agent, mais cela n'entraîne pas de différence notable pour le coût de transport de l'agent, car les classes graphiques sont présentes sur chaque machine Java et ne contenaient que très peu de données. Une troisième version ajoute l'intelligence (les algorithmes de RI) directement dans l'agent. Cela porte la taille des classes à 33.3 Ko, sans compter le grand nombre de connaissances à transporter à chaque déplacement de l'agent, ce qui rend cette option particulièrement inintéressante et justifie le

développement et l'utilisation d'une architecture multi-agents pour garder à la fois l'intelligence et des performances correctes, telles celles présentées dans le Tableau 4.2.

Les versions de l'application utilisées à partir de maintenant sont basées sur l'architecture développée et présentée dans ce mémoire. La différence entre les différentes versions porte sur l'utilisation des connaissances réseau à l'intérieur de la classe *AgentItinerary*. La première version «simple» va parcourir les destinations dans l'ordre de priorité. La version «locale» va chercher à parcourir en priorité les destinations se trouvant dans la zone courante. La version «complexe» va attribuer une priorité à chaque zone connue pour savoir où se rendre ensuite. Chacune de ces versions est plus complexe que la précédente. Elle doit donc être plus «lourde» à transporter. Cependant, on attend de cette complexité une réduction du nombre de voyages nécessaires, et donc une réduction de la charge totale du réseau. La Figure 4.5 illustre les différences entre les deux derniers algorithmes.

Tableau 4.2 Comparaison du coût de transport entre différentes versions

	Sens	Aller (Koctets)	Aller - Retour
1er envoi de « simple »	1->2	18	19
	2->1	3.2	8
	Total	21.2	27
2eme envoi de « simple »	1->2	5	5.7
	2->1	0.5	5.7
	Total	5.5	11.4
1er envoi de «local»	1->2	18	19
	2->1	3	9
	Total	21	28

On remarque dans le Tableau 4.2 qu'il n'y a pas de grande différence entre les différentes versions, tant au niveau du premier envoi que des suivants. D'autres mesures, non exposées ici, ont confirmé ces résultats. De plus, l'utilisation du cache, tant au

niveau de Grasshopper que de la machine Java (chargement de l'objet *Class*), permet de limiter d'éventuelles différences en gardant le code des algorithmes. On peut se demander si la taille de l'agent va beaucoup varier en cours de route, alors qu'il collecte des connaissances et des résultats. Pour vérifier cela, nous avons mesuré la charge d'un trajet (aller) de l'agent avec un plus grand nombre de destinations initiales. Nous n'avons alors mesuré aucune différence notable avec un itinéraire de 20 adresses sur 2 zones, au lieu de 3 adresses sur 2 zones, et une majoration de 2 Koctets pour un itinéraire de 20 adresses et 20 zones. Considérant que c'est un cas extrême compte tenu de l'application développée et du fait que l'on espère trouver la bonne réponse dans les premières adresses, on peut négliger les variations de la taille de l'agent en cours d'itinéraire.

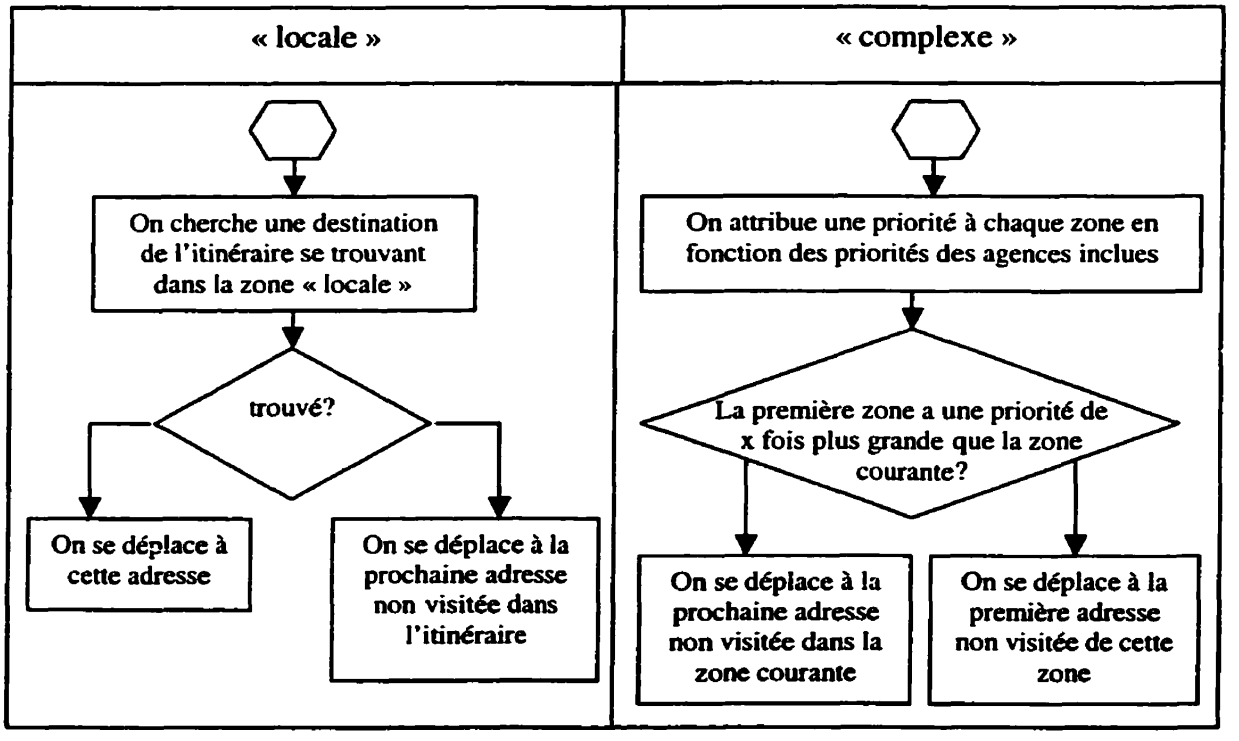


Figure 4.5 Algorithmes de parcours de l'itinéraire

Il faut néanmoins considérer la taille des résultats pour une application chargée de trouver, par exemple, des images, comme l'application de chercheur d'images développée. Dans ce cas, on ne peut plus négliger la taille des images par rapport à la taille de l'agent (10-20 Koctets). Une possibilité est que l'agent ne rapporte alors que des pointeurs vers les images et que celles-ci soient chargées au retour de l'agent sur la machine de l'utilisateur, ou la machine à laquelle il est connecté si la dernière liaison a une grande latence (cas des liaisons sans fil). On n'a alors qu'à ajouter la taille des images pour le dernier trajet ou plus, suivant la configuration du réseau et de l'application (cas où les images doivent traverser plusieurs sous-réseaux avant de parvenir à la machine de l'utilisateur.

4.2.2 Scenarios de recherche d'information

Nous allons comparer les trois algorithmes en simulant leurs déplacements et l'apprentissage sur un scénario regroupant quelques entreprises, administrations et fournisseurs d'accès dans trois villes, par exemple Montréal, Ottawa et Québec. La Figure 4.6 en est une illustration.

Nous avons mesuré le nombre de déplacements des agents pour chaque version pour une succession de requêtes donnée. Nous distinguons les déplacements «locaux», à l'intérieur d'un même sous réseau, des déplacements «régionaux». Les Figures 4.7 à 4.11 montrent l'évolution du nombre de déplacements pour les trois versions de l'algorithme de parcours d'itinéraire développées : «simple», «locale» et «complexe».

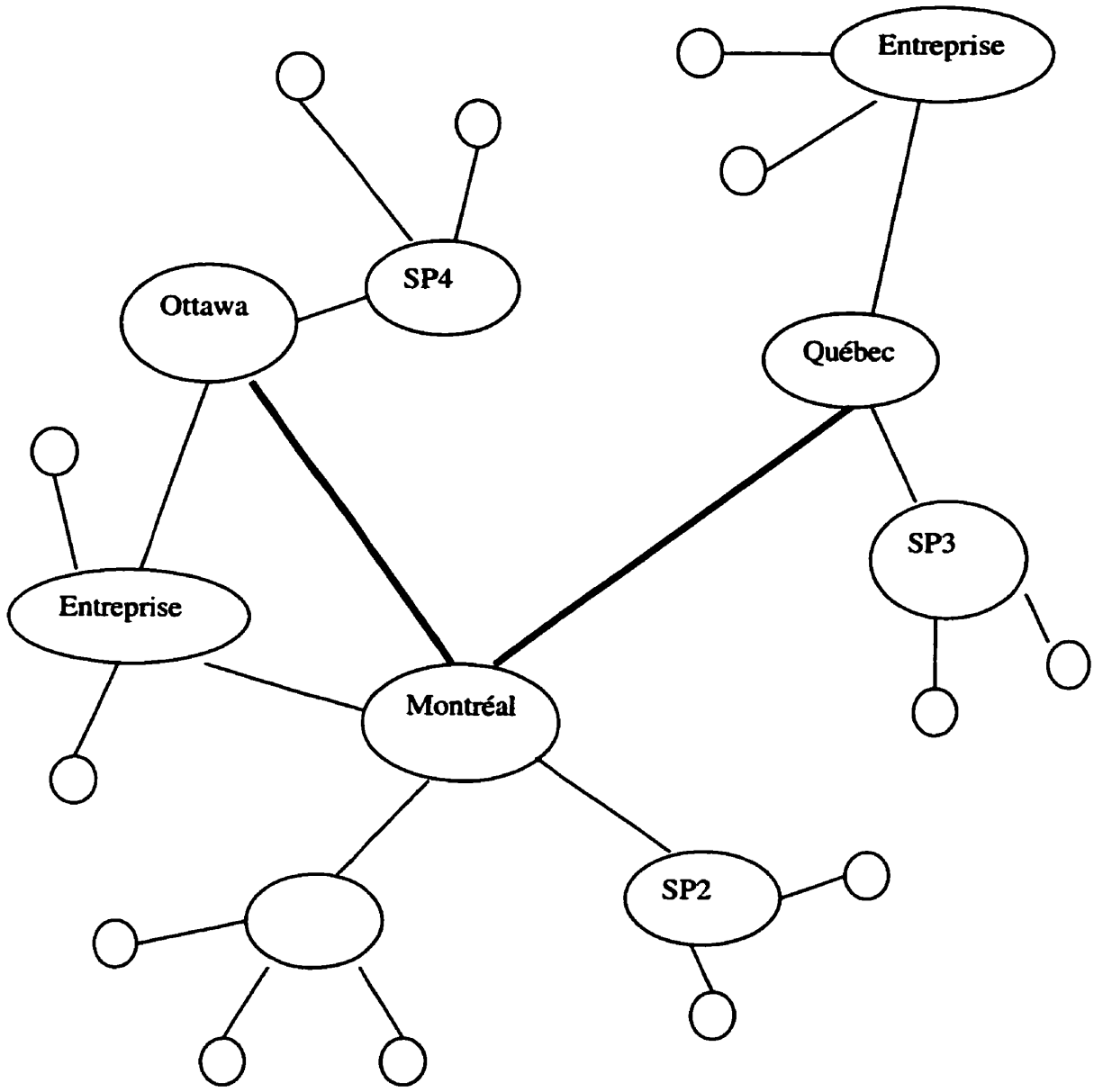


Figure 4.6 Scénario de mesure

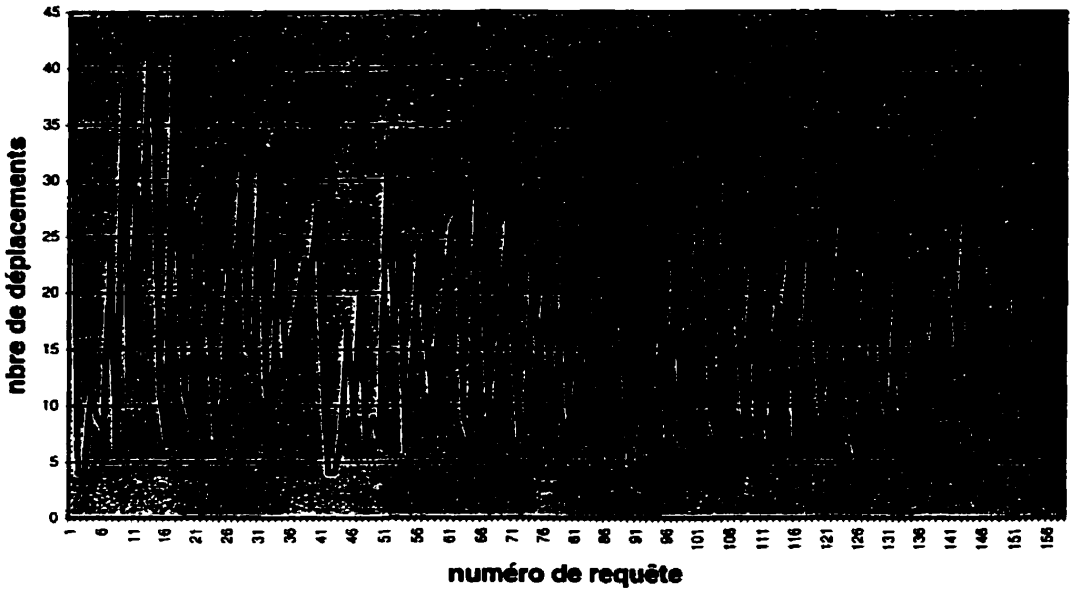


Figure 4.7 Évolution du nombre de déplacements de l'agent «simple»

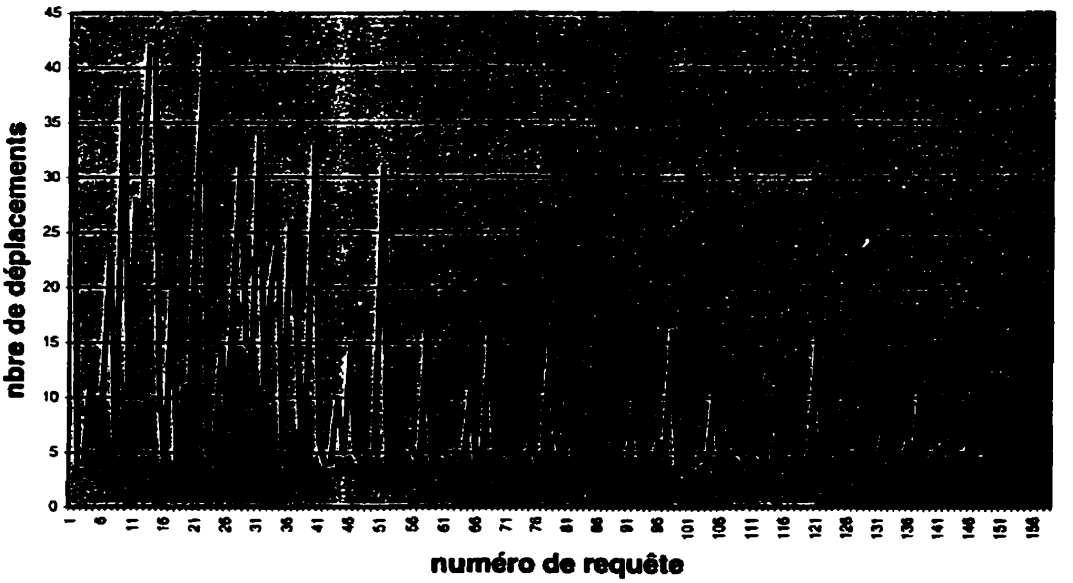


Figure 4.8 Évolution du nombre de déplacements de l'agent «local»

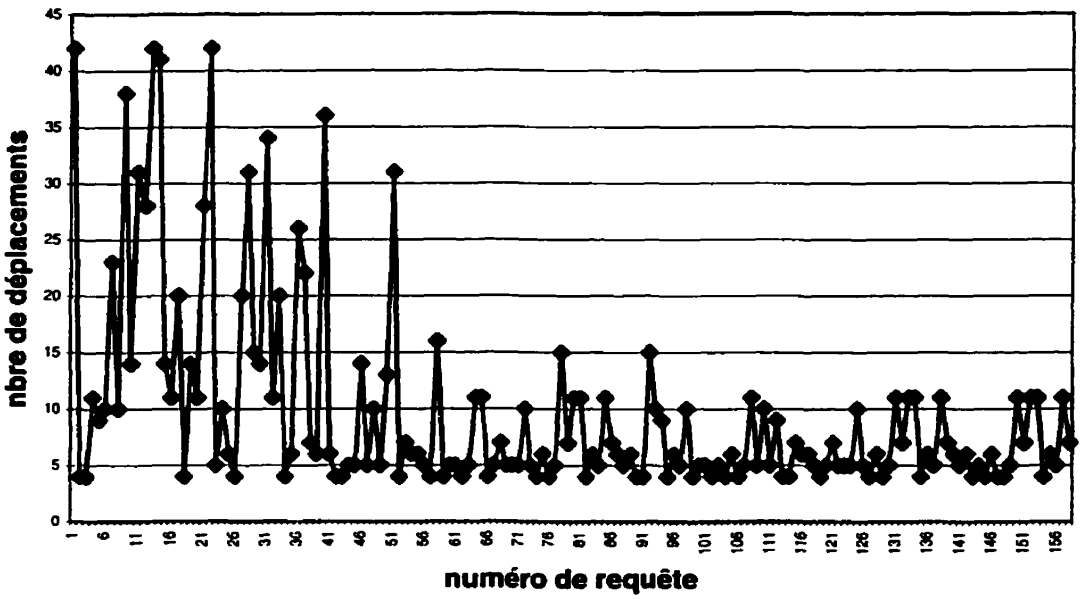


Figure 4.9 Évolution du nombre de déplacements de l'agent «complique»

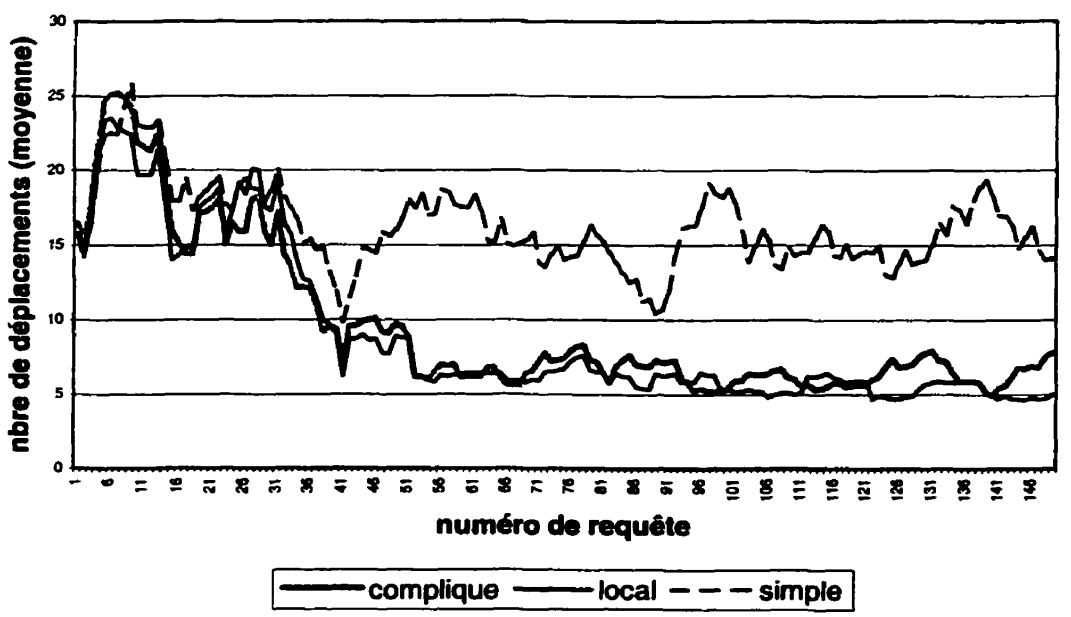


Figure 4.10 Comparaison en moyenne du nombre de déplacements

On constate que les performances de la version «simple» sont les moins bonnes, et celles de la version «compliquée» légèrement moins bonnes que celles de la version «locale». Pour les trois versions, on observe une diminution du nombre de déplacements avec le temps, du fait de l'apprentissage. En effet, les agents donnent des renseignements par « feedback » aux *KnowAgents* qui peuvent ensuite les envoyer plus rapidement au bon destinataire. La différence entre la version simple et les autres peut s'expliquer par le fait que les correspondants cherchés sont souvent dans le voisinage de la machine de l'utilisateur ou d'une machine répondant à une requête proche. Les deux dernières versions tirent parti de ce fait, alors que la version « simple » est perdue des qu'elle ne trouve pas exactement ce qu'elle cherche du premier coup. Cette différence devrait progressivement s'atténuer avec le nombre de requêtes, mais on voit ici que ce processus est très lent pour la version «simple». Les performances de la version complexe sont un peu décevantes, compte tenu de sa plus grande «complexité». La raison en est que le scénario considéré est très simple, et que la version locale trouve facilement ce qu'elle cherche. Les algorithmes plus compliqués ne sont alors pas nécessaires et sont même gênants car plus sensibles à la taille des zones (dans notre cas), alors que la version locale va directement à la machine la plus cotée, qui est généralement la bonne. Il n'a pas non plus besoin de chercher plusieurs résultats, ce qui aurait pu inciter à cibler une zone, plus qu'une seule machine. Il peut alors être intéressant de faire varier la taille du réseau de test et le nombre de machines dans les mesures. Les Figures 4.11 et 4.12 donnent l'évolution et la comparaison de la moyenne du nombre de déplacements pour une zone et pour 5 zones.

On voit que, plus le nombre de zones augmente, plus la différence entre la version «simple» et les autres s'accroît, et moins il y a de différence entre la version «local» et la version «complexe». Ceci corrobore l'analyse des premiers résultats. On peut remarquer de plus que la version «complexe» surpasse la version «locale» à la fin de la phase d'apprentissage, au moment où il commence à y avoir plus de connaissances, mais pas suffisamment pour que les algorithmes fonctionnent de façon «optimale». Cela recommande la version « complexe » pour les environnements réels et dynamiques.

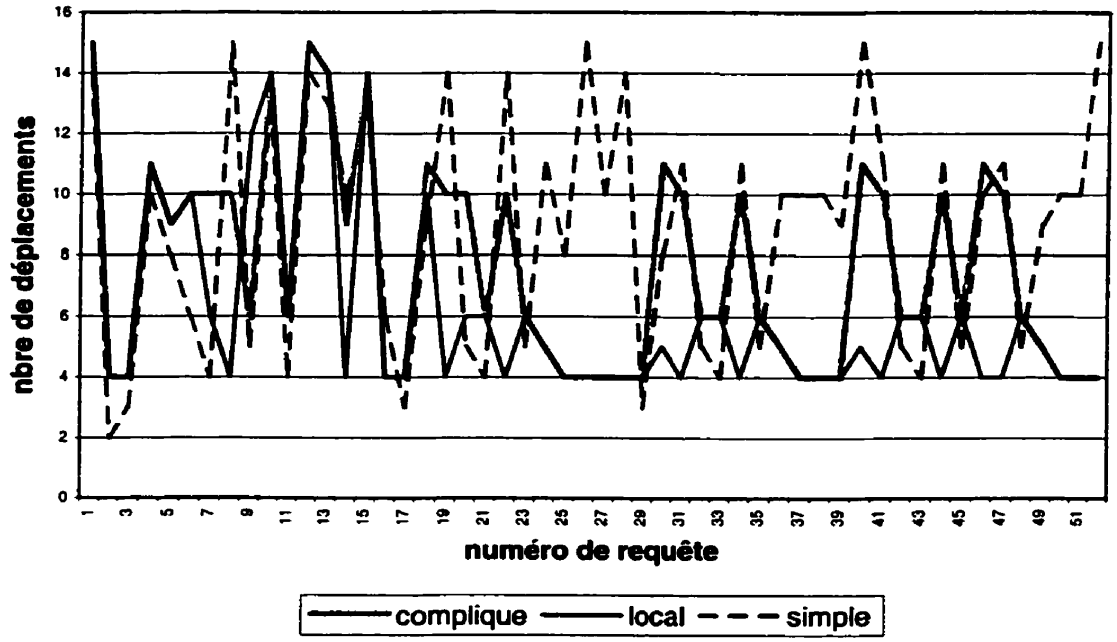


Figure 4.11 Moyenne du nombre de déplacements pour 1 zone

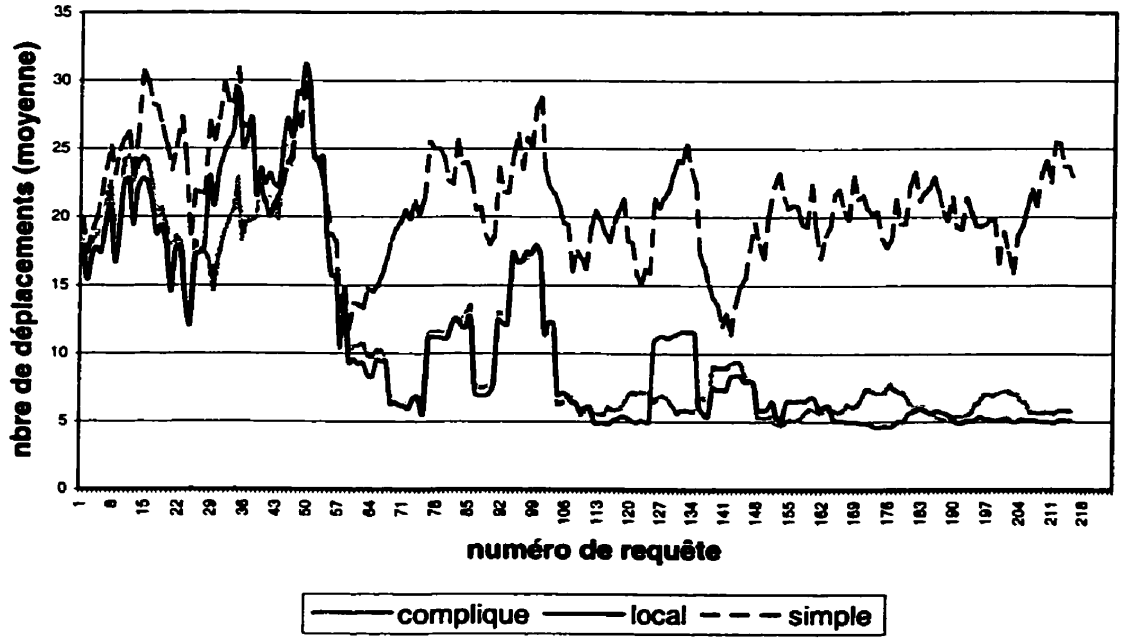


Figure 4.12 Moyenne du nombre de déplacements pour 5 zones

On peut noter que 4 est la valeur inférieure limite pour ces mesures. Cela suppose que l'agent se rend du terminal de l'usager à l'agent de recherche d'information puis directement à la bonne destination, et prend le chemin inverse pour revenir afin de donner son « feedback » au *KnowAgent*. L'évolution du nombre de déplacements « régionaux » est donnée par la Figure 4.13.

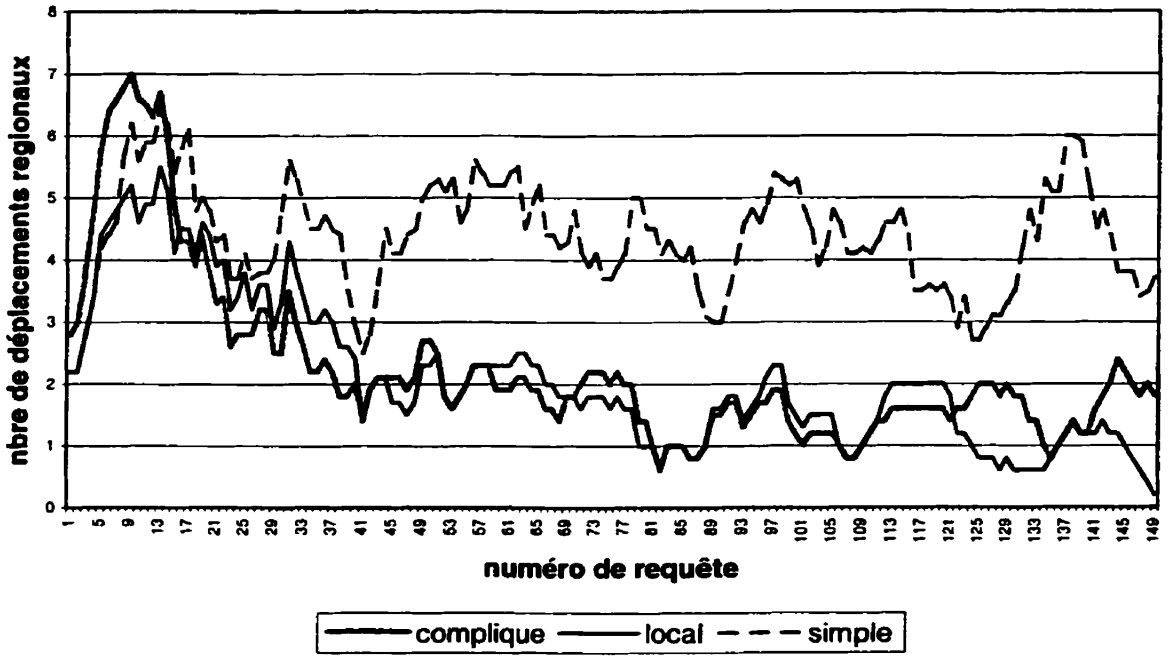


Figure 4.13 Évolution du nombre de déplacements « régionaux »

Comme pour le nombre total de déplacements, le nombre de déplacements «régionaux» diminue avec l'apprentissage, particulièrement pour les deux dernières versions, car l'information voulue est ramenée au niveau local. Celles-ci présentent des performances équivalentes, avec toutefois une légère supériorité de la version «complexe» pendant la phase d'apprentissage qui s'inverse par la suite, pour les mêmes raisons que le nombre total de déplacements. Dans ces mesures, on se limite également à un seul niveau de zones, alors qu'on pourrait tirer parti de plusieurs pour que les agents fassent une différence entre les déplacements entre villes et entre sous-réseaux.

Une autre constatation faite à la suite de ces mesures est que l'architecture développée rapproche l'information de l'usager. Plus précisément, les agents rapportent l'information voulue au *KnowAgent* de leur zone, grâce au mécanisme de «feedback». Ce fait est appuyé par la mesure du nombre de déplacements effectués lors du retour de l'agent présenté à la Figure 4.9. En effet, l'agent mobile, une fois l'interlocuteur trouvé, s'arrête sur chaque *KnowAgent* contacté pour lui présenter ses résultats en vue d'apprentissage. Le nombre de déplacements effectués lors du retour de l'agent correspond donc au nombre de *KnowAgent* contactés lors du trajet aller plus un. On constate que, une fois la phase d'apprentissage passée, l'agent ne contacte souvent qu'un seul *KnowAgent*, celui de sa zone, par lequel il commence tout trajet. L'information a donc été rapprochée de l'utilisateur, ce qui est un plus dans l'utilisation des ressources réseau. Cela est possible dans ce scénario du fait de la petite taille du réseau considéré. Chaque *KnowAgent* peut contenir les informations de tout le réseau. Ce ne serait pas possible avec un réseau plus grand. La connaissance se répartirait alors entre les différents *KnowAgent*, d'où l'intérêt théorique d'une recherche par zones.

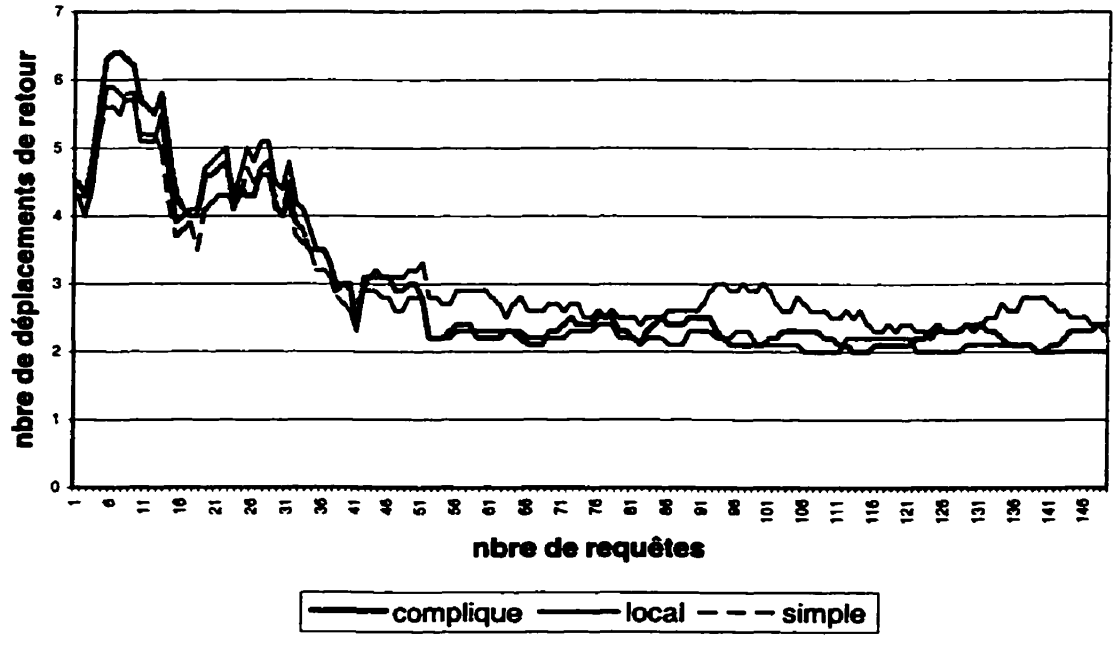


Figure 4.14 Nombre de déplacements au retour

On peut également chercher à mesurer d'autres aspects de l'architecture à travers un scénario plus complexe. Alors que les connexions sont considérées statiques dans le premier scénario et que les coûts dus aux chargements des services sont négligés, il serait plus réaliste, surtout dans le domaine des télécommunications sans fil où la durée des connexions est plus réduite, de les considérer dynamiques. On reprendrait alors le même scénario que précédemment, mais en considérant que les utilisateurs et les serveurs se connectent et se déconnectent régulièrement. Une application revenant sur ces serveurs les obligerait donc à recharger les services dont elle a besoin. Cela ferait intervenir les mécanismes d'adaptation de l'architecture et de migration des services qui sont « ignorés » dans le premier scénario. Toutefois, ces coûts concernent surtout les serveurs, et non les terminaux usagers qui n'en contiennent que quelques-uns. Nous avons donc décidé, pour des raisons de restriction de temps, de ne pas implémenter un tel scénario.

On peut constater que, d'après ces mesures, on a effectivement gagné en intelligence et en utilisation des ressources réseau grâce à l'architecture multi-agents développée et à l'utilisation d'algorithmes de routage tirant profit de connaissances sur la topologie du réseau, par rapport à l'implémentation initiale. Une implémentation client serveur est toutefois, pour cette application, moins coûteuse (l'établissement d'une communication par SIP ne prend que 500 octets environ), mais cette implémentation est moins personnalisable et moins souple. Elle est aussi moins évolutive, car tous auraient à passer par un serveur central basé dans une seule ville, ce qui augmente le nombre de connections et de données «régionales». Une implémentation client/serveur distribuée serait la plus performante, mais reste moins personnalisable et moins souple qu'une implémentation agent mobile.

Chapitre V
Conclusion

La technologie *agent mobile* est apparue en 1995 comme une extension du code mobile et de la migration de processus. Elle doit, de par sa conception, réduire la quantité de données transmises à travers un réseau pour une même tâche effectuée à distance par rapport à la technologie client/serveur dans certaines circonstances. Cependant, les mesures de comparaison de performances entre la technologie *agent mobile* et la technologie *client/serveur* effectuées par le passé ne sont pas convaincantes. Souvent, on peut déplorer un scénario par trop favorable à l'agent ou une implémentation du serveur déficiente, voire inexistante. Il faut dire aussi que l'agent mobile souffre souvent de devoir transporter tout le code dont il a besoin ou ne bénéficie pas, sur les serveurs, d'une interface adaptée. L'architecture multi-agents présentée dans ce mémoire vise à résoudre ce problème en fournissant aux agents mobiles un moyen d'obtenir les services et l'interface dont ils ont besoin sur chaque machine. Elle aborde aussi le problème de la représentation et du partage des connaissances des agents sur leur environnement. Les mesures réalisées montrent que cette architecture atteint ses buts. Même si la comparaison avec une architecture client/serveur peut paraître défavorable, il faut penser que la technologie agent mobile ne vise pas à se comparer à la technologie client/serveur en terme de performances pures, mais à éviter ses défauts comme le manque d'évolutivité ou de souplesse.

5.1 Synthèse des travaux et contributions principales

Nous avons défini et implémenté une architecture multi-agents mobiles de recherche d'information au travers de deux applications, le «HuntGroup», un service avancé de téléphonie de recherche de correspondant, et un agent chercheur d'images sur Internet.

Cette architecture vise en premier lieu à mettre en relation un agent mobile arrivant sur une machine avec les agents se trouvant sur cette même machine capables de lui fournir les services dont il a besoin. Elle fait correspondre un service avec une interface Java particulière correspondant à ce service. Ainsi, les agents pourront ensuite utiliser cette interface pour établir un «proxy» de communication. Cela permet d'avoir un cadre précis faisant le lien entre les classes Java d'un agent et l'ensemble des fonctions constituant un service. Pour pouvoir mettre les agents en relations de la façon la plus souple possible, nous avons implémenté notre propre fonction de recherche de services dans un agent particulier de notre architecture, le *Registraire*. Celui-ci est en quelque sorte le prolongement de la plate-forme d'agents mobiles. Le système d'agents mobiles utilisé, Grasshopper, offre déjà des fonctions de recherche d'agents sur différents critères qui restent utilisables, mais aucune n'était suffisamment souple, ni ne permettait à un seul agent d'offrir plusieurs interfaces. De plus, le registraire se charge de chercher un service sur les machines voisines et dans tout le réseau, puis de le charger s'il ne se trouve pas localement. Ceci nous amène au concept de serveur dynamique dans lequel les services et l'interface présentés par un serveur peuvent changer dynamiquement selon la demande des applications l'utilisant et les règles et limitations du serveur. La demande des applications est représentée par le chargement de services par le registraire sur demande des agents. Les règles et limitations du serveur peuvent être traduites au niveau du registraire, du système d'agents mobiles, des deux, ou par un autre agent. Plus le serveur offre une interface initiale de bas niveau, plus les possibilités de modification et d'optimisation en fonction de chaque service utilisateur sont grandes.

Un autre aspect abordé est celui de la représentation des connaissances sur l'environnement des agents, en particulier la topologie du réseau et les autres agents. L'idée de voisinage du registraire est traduite en zones à l'échelle du réseau entier et d'une relation d'inclusion. Cela permet aux agents d'avoir une notion de « proximité » entre deux machines et d'en tenir compte au niveau des algorithmes de routage. Les autres connaissances sont traitées dans notre implémentation par des algorithmes de recherche d'information vectoriels et de «feedback».

Des mesures de performance ont permis de prouver la validité de la conception de l'architecture. Nous nous sommes focalisés sur l'apport effectif de l'utilisation de connaissances sur le réseau et d'algorithmes de routage plus complexes au regard de l'augmentation de la taille de l'agent qu'ils entraînent. Nous avons constaté que, du fait de l'utilisation du cache de Grasshopper et de la machine Java, le déplacement d'un agent utilisant l'un ou l'autre des algorithmes est équivalent. Considérant maintenant le nombre de déplacements de l'agent nécessaires pour trouver le bon correspondant avec chaque algorithme, tous mettent à profit le mécanisme d'apprentissage par rétroaction pour réduire leurs déplacements. Les deux algorithmes utilisant en plus des connaissances sur la topologie du réseau montrent un réel avantage. Cependant, les mesures n'ont pas réussi à mettre en évidence une supériorité de l'algorithme le plus complexe par rapport au moins complexe parmi ces deux algorithmes.

5.2 Limitations et recherches futures

L'architecture décrite dans ce mémoire a été plus particulièrement étudiée et implémentée pour les applications de recherche d'information. Elle peut nécessiter de légères modifications pour être utilisée par toute sorte d'applications, notamment au niveau des interfaces. L'implémentation de communications KQML peut résoudre ce problème en apportant plus de souplesse. Une partie de l'architecture, notamment les fonctions d'administration de serveur, restent également à implémenter. Les aspects de sécurité ont été également très peu abordés. L'architecture et les algorithmes exposés n'ont pas été testés en condition réelle. On a vu que cela pouvait limiter les capacités d'apprentissage et donc les performances des algorithmes de routage et de recherche d'information. L'architecture même n'a pas été extensivement testée, notamment en ce qui concerne les mécanismes de recherche et de chargement des services, ou encore l'administration de serveur et le support de nombreux agents.

Les recherches futures devraient s'orienter vers l'implantation de mécanismes de communications par messages KQML, qui devraient donner beaucoup plus de souplesse

à l'architecture développée. Les aspects de sécurité sont également nombreux, que ce soit au niveau de la protection des données sensibles ou de l'élimination de services et d'agents malveillants ou défectueux. De plus, même si l'architecture proposée permet de réduire considérablement la charge du réseau par rapport à une application agent mobile classique, ses performances restent faibles en comparaison avec une application client/serveur du fait de la taille importante des données transportées. Une voie intéressante pour résoudre ce problème est celle suivie par D. B. Lange (site Lange), un des pères des systèmes agents mobiles, qui développe des agents mobiles en XML. Le fait de mettre les données en format texte plutôt que sous forme de classes Java sérialisées pourrait diminuer leur taille et les rendre directement utilisables par des agents utilisant des classes ou un langage différent, mais nécessite de surcharger les fonctions de sérialisation et désérialisation au niveau de l'agent, mais aussi de la plate-forme.

Toutefois, l'architecture proposée ici offre une possibilité intéressante d'intégration de systèmes d'agents intelligents et mobiles. De plus, les mesures effectuées montrent l'intérêt des algorithmes de recherche d'information et de routage utilisés.

Bibliographie

S. Abu-Hakima, R. Liscano and R. Impey, "A common Multi-agent Testbed for Diverse Seamless Personal Information Networking Applications", *IEEE Communications, Special Issue on Mobile Agents and Telecommunications*, vol. 36, no. 7, 1998, pp. 68-74

Y. Aridor and D. B. Lange, "Agent Design Patterns: Elements of Agent Applications Design", *Proc. Of the Second International Conference on Autonomous Agents (Agents '98)*, Mai 1998, pp. 108-15.

Y. Aridor and M. Oshima, "Infrastructure for Mobile Agents: Requirements and Design", Proc. of 2nd International Workshop on Mobile Agents (MA '98), Springer Verlag, Berlin, Septembre 1998, pp. 38-49.

J. Baumann, F. Kohl, K.Rothermel and M.Strasser, "Mole – Concepts of a Mobile Agent System", *Mobility Processes, Computer, and Agents*, ACM Press, Addison-Wesley, Reading , 1998, pp. 461-492.

L. F. Bic, M. B. Dillencourt, J. M. Cahill, M. Fukuda, "Messages versus Messengers in Distributed Programming", *Journal of Parallel and Distributed Computing*, vol. 57, no. 2, 1999, pp. 188-211.

B. Brewington, robert Gray, K. Moizumi, D. Kotz, G. Cybenko, D. Rus, « Mobile Agents in distributed information retrieval », Matthias Klusch, (ed.), *Intelligent Information Agents*, chapter 15, Springer-Verlag, Berlin, 1999.

S. Derochers, R. Glitho, K. Sylla, "Experimenting with PARLAY in a SIP Environment: Early Results " IPTS 2000, Sept 11 2000, Atlanta, GA, USA.

F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *Mobility Processes, Computer and Agents*, ACM Press, Addison Wesley, Reading, 1998, pp. 57 – 86.

B. Emako-Lenou, « Agent-based Technologies: Concepts and Applications", Mémoire de maîtrise, département de genie électrique et genie informatique, École Polytechnique de Montréal, février 2000.

L. Gannoun, J. Francioli, S. Chachkov, F. Schutz, J. G. Hulaas, and J. Harms, "Domain Name eXchange : A Mobile-Agent-Base Shared registry System", *IEEE Internet Computing*, mars/avril 2000, pp. 59-64.

R.S. Gray, "PhD Thesis Proposal : Transportable Agents", Dept. of computer science, University of Dartmouth, mai 1995.
http://www.cs.dartmouth.edu/reports/abstracts/TR95-261/

R. S. Gray, D. Kotz, G. Cybenko, D. Rus. « D'Agents: Security in a multiple-language, mobile-agent system », Giovanni Vigna, editor, *Mobile Agents and Security, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1998.

K. Hafner, "Have your agent call my agent", Newsweek, 75(9), Février 1995.

C. G. Harrison, D. M. Chess, A. Kershenbaum, "Research Report : Mobile Agents: Are they a good idea?", 28 mars 1995.

C. Hewitt, « Viewing Control Structures as Patterns of passing messages », *Journal of Artificial Intelligence*, 8, June 1977, pp 323-364.

G. Karjoth, D. B. Lange, and M. Oshima, "A Security Model for Aglets", *IEEE Internet Computing*, vol. 1, no. 4, July/August 1997, pp. 68-77.

A. Karmouch and V. A. Pham, "Mobile Software Agents: an Overview", *IEEE Communications, Special Issue on Mobile Agents and Telecommunications*, vol. 36, no. 7, 1998, pp. 26-37.

D. B. Lange, « Mobile Objects and Mobile Agents: The Future of Distributed Computing », Proceedings of The European Conference on Object-Oriented Programming '98, Brussels, 1998, pp. 1-12.
http://www.acm.org/~danny

D. Lange and M. Oshima,"Programming and Deploying Java Mobile Agents with Aglets", The Aglet book, Addison-Wesley, Reading.

T. Lindholm, F. Yellin, « The Java Virtual Machine Specification », Addison-Wesley, Reading, 1996.

D.S. Miljicic and al., "MASIF, The OMG Mobile Agent System Interoperability Facility", Proceedings of the Second International Workshop on Mobile Agents, septembre 1998, pp. 50-67.

W. Moussawi, «Modélisation d'un agent de recherche intelligente d'information sur Internet», mémoire de maîtrise, Dpt. de génie électrique et génie informatique, École Polytechnique de Montréal, novembre 2000.

G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa, "A flexible and efficient Java environment mixing bytecode and compiled code", *Proceedings of Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, 1997, pp. 1-20.

S. Myaeng, R. R. Korfhage, „Integration of user profiles : models and experiments in information retrieval". *Information processing and management*, vol. 26, ISSN 0306-4573, avril 1990, pp. 719-738

N.J. Nilsson, "Learning Machines - Foundations of Trainable Pattern Classifying Systems", McGraw-Hill, NewYork, 1965.

P. Noriega Blanco Vigil, "Agent Mediated Auctions: The Fishmarket Metaphor", PhD, 1997.

H. Ouahid, A. Karmouch, "An XML-Based Web Mining Agent", in *Proceedings of MATA'99*, pp. 393-404.

M. Powell, B. Miller, "Process Migration in DEMOS/MP", in *Mobility Processes, Computer and Agents*, ACM Press, Addison Wesley, Reading, 1998, pp. 29–38.

M. Ranganathan, A. Acharya, S. Sharma et J. Saltz, "Network-aware Mobile Programs", in *Mobility Processes, Computer, and Agents*, ACM Press, Addison-Wesley, Reading, 1998, pp. 461-492.

J.J. Rocchio, "Document retrieval systems - Optimization and evaluation", Ph.D. Thesis, Harvard University, Report ISR-10 to National Science Foundation, Harvard Computation Laboratory (1966).

A. Rogers, "Is there a case for viruses?", Newsweek, 75(9), February 27, 1995.

G. Salton, A. Wong, S. Yang, «A Vector Space Model for Automatic Indexing». *Communications of ACM*, vol. 18, no. 11, 1975, pp. 613-620.

G. Salton, M. McGill, "Introduction to modern information retrieval", *Computer Science Series*, McGraw Hill Book Company, new-York, ISBN 0-07-054484-0, AACR2, 1983.

T. Sandholm, Q. Huai, "Nomad : Mobile Agent System for an Internet-Based Auction House", *IEEE Internet Computing*, Mars/Avril 2000, pp. 74-79.

Y. Shoham, "An overview of agent-oriented programming", J. M. Bradshaw editor, Software Agents, MIT press, 1997, pp 271-290.

A. di Stefano, C. Santoro, "NetChaser : Agent Support for Personal Mobility", *IEEE Internet Computing*, Mars/Avril 2000, pp. 74-79.

N. Wiener, "Cybernetics or Control and Communication in the Animal and Machine", MIT Press, Cambridge, 1948.

M.J. Woolridge, N. R. Jennings, "Software Engineering with Agents : Pitfalls and Pratfalls", *IEEE Internet Computing*, May-June 1999, p. 20-27.

"Mobile Agents in the Context of Competition and Cooperation (MAC3), a workshop at Autonomous Agents '99", May 1 1999, Seattle, Washington, USA http://mobility.lboro.ac.uk/MAC3/

Sites Internet

ActComm	http://actcomm.thayer.dartmouth.edu/
Aglets	http://www.trl.ibm.co.jp/aglets/
Alexa	http://www.alexa.com
auction	http://auction.eecs.umich.edu
Auctionet	http://www.auctionet.com
Cetus-links	http://www.cetus-links.org/oo_mobile_agents.html
Concordia	http://www.meitca.com/HSL/Projects/Concordia/
D'Agents	http://agent.cs.dartmouth.edu/
Dartmouth-réseau	http://agent.cs.dartmouth.edu/network/index.html
DNX	http://cui.unige.ch/DNX
Ebay	http://www.ebay.com/aw
Ewatch	http://www.ewatch.com/
excite	http://live.excite.com/
Frictionless	http://www.frictionless.com

JavaSpace	http://java.sun.com/products/javaspaces/
Jorstad	http://agent.cs.dartmouth.edu/workshop/1997/slides/jorstad/
KSE	http://www.cs.umbc.edu/kse/
Lange	http://www.acm.org/~danny
Lange-raisons	http://www.cs.dartmouth.edu/~dfk/papers/kotz:future2/#lange:reasons
Linda	http://www.cs.yale.edu/Linda/linda.html
liste :	http://mole.informatik.uni-stuttgart.de/mal/preview/preview.html
Loughborough	http://luckyspc.lboro.ac.uk/Docs/index.html (page inaccessible)
Messengers	http://www.ics.uci.edu/~bic/messengers/
Mole	http://mole.informatik.uni-stuttgart.de
Objectspace	http://www.objectspace.com/
Onsale	http://www.onsale.com
Parlay	www.parlay.org
Recherche-IBM	http://www.research.ibm.com/massive/
Sumatra	http://www.cs.umd.edu/~acha
Tacoma	http://www.tacoma.cs.uit.no/
Zdnet	http://www.zdnet.com/zdi/pview/pview.cgi