Titre: Title:	Services sur réseaux mobiles : architecture d'agent et maintenance
Auteur: Author:	Bertrand Emako Lenou
Date:	2000
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Emako Lenou, B. (2000). Services sur réseaux mobiles : architecture d'agent et maintenance [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8628/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8628/
Directeurs de recherche: Advisors:	Samuel Pierre, & Roch Glitho
Programme: Program:	Génie électrique

UNIVERSITÉ DE MONTRÉAL

SERVICES SUR RÉSEAUX MOBILES : ARCHITECTURE D'AGENT ET MAINTENANCE

EMAKO LENOU

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)

(OCTOBRE 2000)

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

SERVICES SUR RÉSEAUX MOBILES :

ARCHITECTURE D'AGENT ET MAINTENANCE

Présenté par : EMAKO LENOU

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

A été dument accepté par le jury d'examen composé de :

Michel Dagenais, Ph.D.	Président
Samuel Pierre, Ph. D.	Directeur
Roch Glitho, M. Sc. A.,	Co-Directeur
Alejandro Quintero, Ph. D.	Membre

REMERCIEMENTS

Pour réaliser ce mémoire, j'ai bénéficié de l'aide de plusieurs personnes. Mes sincères remerciements à tous ceux qui directement ou indirectement ont contribué à ce mémoire.

Mes parents qui m'ont toujours supporté et ont enduré beaucoup de privations pour me permettre de réussir.

Mes encadreurs Roch et Samuel.
Roch est l'initiateur et directeur du projet. C'est sous son tutelage que j'ai compris la téléphonie Internet, les agents mobiles, le processus et la rigueur de la recherche scientifique. Il est exceptionnel d'avoir un encadreur éminemment qualifié, disponible, qui évalue sans complaisance et expose ses étudiants aux meilleures avancées technologiques.

Rien n'aurait été possible sans Samuel. En mettant ses étudiants en contact avec le milieu industriel, il leur rend un service inestimable. Certains écueils ne sont pas reliés à la recherche elle-même, la médiation de Samuel m'a été précieuse pour les surmonter.

Tous les deux, ils m'ont aidé à démarrer ma carrière, plutôt qu'à simplement achever ma maîtrise.

RÉSUMÉ

La téléphonie Internet offre beaucoup d'opportunités. Les coûts sont réduits et des services novateurs peuvent être offerts sur le marché. Des critères ont été formulés pour évaluer les architectures de services pour la téléphonie Internet. Ces critères incluent entre autres l'accès universel, la possibilité de personnaliser les services et la création et le déploiement rapide des services. L'accès universel implique que les services doivent être disponibles quelque soit les différentes machines que l'usager utilise et leur position par rapport aux zones de couverture du réseau. La personnalisation signifie que l'abonné peut configurer le comportement de ses services et, ou imbriquer l'exécution de services différents pour en créer de nouveaux qui correspondraient à ses préférences. Les autres critères sont l'indépendance des services et du réseau, une gestion simple des services, l'interopérabilité avec les autres architectures de services et le support pour une large palette de services.

Les architectures de services pour la téléphonie Internet basées sur les agents ont été proposées récemment. Elles préconisent l'utilisation d'agents mobiles qui sont semblables à des conteneurs en cela qu'ils transportent les exécutables des services (ou les pointeurs aux exécutables). Transporter des exécutables (ou des pointeurs) amène de nouveaux défis. Le traitement des souscriptions est un d'entre-eux. Ce mémoire traite de ces architectures à base d'agents puis propose et évalue des alternatives de traitement des abonnements.

Le cycle de vie d'un service est composé de sa création, son déploiement, son utilisation et son retrait. La souscription est la principale difficulté du déploiement et du retrait des services. L'agent mobile qui transporte des services doit être mis à jour quand un abonné souscrit à un ou plusieurs nouveaux services (ou nouvelles versions de services existants). Il y a deux approches possibles : la permutation d'agent et la mise à jour dynamique de l'agent. Dans la première approche, l'agent qui transporte les services est remplacé par un nouvel agent qui contient les anciens (nouvelles versions au besoin) et les nouveaux services. Selon la seconde approche, les nouveaux services sont insérés

dynamiquement dans l'agent et les anciennes versions des services sont dynamiquement changées pour refléter les nouvelles versions. La permutation d'agents a deux variantes, la permutation progressive et la permutation abrupte. La mise à jour dynamique est radicalement différente des techniques traditionnelles de mise à jour. D'ordinaire, l'application est arrêtée, désinstallée, puis la nouvelle version installée et dans certains cas la machine est réinitialisée. Ceci occasionne un temps mort inacceptable en téléphonie. La mise à jour dynamique insère la nouvelle version d'un service dans ledit service pendant qu'il est en cours d'exécution, sans qu'il n'y ait d'interruption de service.

Les requis pour les mises à jour incluent la minimisation de la durée d'interruption des services, la validité des changements et l'extensibilité à un nombre élevé de services. Toutes les approches proposées sont extensibles comme le prouvent les résultats de mise en oeuvre des prototypes implémentés. Par contre, la permutation abrupte provoque une très brève interruption de service au contraire de la permutation progressive et la mise à jour dynamique n'en causent pas.

Notre implémentation est la preuve du concept d'architecture d'agent pour services sur réseaux mobiles. Nous présentons la première spécification technique et la première implémentation de l'architecture. Ensuite, nous abordons la gestion du cycle de vie des services. Les solutions possibles (permutation, mise à jour dynamique) sont nouvelles. Le paradigme de mise à jour dynamique que nous avons développé est plus efficient que les systèmes de mise à jour dynamique proposé jusqu'à présent. En ce regard, il se revèle être une contribution importante dans le domaine de l'évolution dynamique des systèmes. Son implémentation résulte en une librairie compacte de 9 KiloOctets. Il est le premier système de mise à jour dynamique qui soit applicable sur les moniteurs d'information portables, i.e. PalmTop, Laptop, Wap Phone, etc.

Notre contribution suivante consiste en une évaluation de performance du paradigme agents mobiles. Cette évaluation détaillée prend en compte les divers facteurs influant sur la performance du code mobile et permet de recueillir des données pertinentes qui font souvent défaut quand on veut procéder à une évaluation rigoureuse des agents mobiles. Elle valide aussi le concept, car les résultats démontrent que la

performance du système reste bonne tout en préservant les avantages inhérents aux agents (accès universel, personnalisation, intelligence, etc.). Nous procédons aussi à une évaluation de performance de nos schémas de mise à jour. Cette évaluation constate que la dégradation de performance dû au mise à jour est négligeable. Plus important encore, dans le cas des mises à jour dynamique, l'utilisation des classes dynamiques n'inflige pas de délai qui puisse être remarqué interactivement.

ABSTRACT

Internet telephony brings a host of opportunities. Cost can be reduced and new, unforeseen services can be engineered. Internet telephony service architectures should provide universal access, support for a wide range of services, tailored services, service and network independence, multi-player environment, rapid service creation and deployment, service manageability and interwork with other service architectures. Mobile agent based service architectures for Internet telephony, have emerged in the recent past. They stipulate the use of mobile agents that act as folders and carry the executables of services (or pointers to the executables). Carrying executables (or pointers) in a mobile agent brings new challenges. Subscription handling is among them. This thesis dissects the architecture, then proposes and evaluates subscription handling alternatives.

Service life-cycle is made of creation, deployment, utilization and withdrawal. Subscription is the core difficulty of service deployment and withdrawal. When the user subscribes to new services (or new versions of existing services), mobile agents that carries services must be upgraded. There are two approaches : agent swapping and on-the-fly updating. In the first approach, the agent that carries the services is swapped with a new agent that carries both the old and the new services. In the second approach, the new services are inserted in the agent on the fly and the old versions of existing services are dynamically changed to reflect the new version. Swapping has two variants *smooth swapping and abrupt swapping*. Upgrade requirements include minimal service interruption, scalability, validity of changes. The solutions proposed, scale and there is either no service interruption (smooth swapping, dynamic update) or an insignificant one (abrupt swapping) as shown by the prototyping results. The dynamic update solution is novel and is applicable to any software program written in Java. It enables the selection of the instances to update and the specification of an adaptable update policy between class versions. Changes are introduced through dynamic Java Classes (granularity). This thesis provides the first experimental, hard evidence of the viability of the architecture.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABBRÉVIATIONS

Sigle ou abbréviation	Signification
ACL	Agent Communication Language
ACP	Agent Communicateur Personnel
AEE	Agent Execution Environment
AMS	Agent mobile pour Services
APD	Appel de procédures à distance
CNRC	Conseil National de Recherche du Canada
EDI	Electronic Data Interchange
ITI	Institut pour les Technologies de l'Information
JDK	Java Development Kit
JKP	Java KIF Parser
JKQML	Java KQML
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
UGS	Unité de Gestion des Services
UPS	Unité de Publication des Services
UCS	Unité de Création des Services
SIP	Session Initiation Protocol
SSL	Secure Socket Layer
SBC	Système à Base de Connaissances

CHAPITRE I

INTRODUCTION

Les architectures distribuées sont devenues récemment le centre d'intérêt de la recherche sur les infrastructures logicielles, notamment à cause de l'Internet. Le slogan "le réseau c'est l'ordinateur" se matérialise et la réseautique influence maintenant le design et l'implémentation de toutes les infrastructures logicielles, des systèmes d'exploitation aux langages de programmation en passant par les applications. De plus, les services sont maintenant disponibles en location à travers l'Internet et cette approche s'illustre comme la voie de l'avenir. On paye pour une certaine durée et pendant cette période on peut accéder le site Web de la compagnie et exécuter les services pour lesquels on a un abonnement. Pas besoin d'installation et les mises à jour sont automatiques (voir site MSNBC). La majorité des produits informatiques devrait évoluer pour devenir de tels services. Toutefois, ces fonctionnalités demandent des connexions réseaux à haute vitesse. Or, la prolifération des moniteurs d'information portables (ordinateur personnel, "palmtop", etc.) qui disposent seulement d'une faible bande passante, impose la recherche d'une nouvelle *ingénierie des services*. Une des approches proposées par l'informatique mobile est la mobilité du code. L'exemple le plus courant de code mobile est les "applets" Java exploités dans les fureteurs (Mosaic, Netscape, etc.). Ce mémoire propose d'utiliser une entité plus effective de code mobile, i.e. les *agents mobiles* pour trouver une solution aux problèmes de la mobilité du couple service-usager et de la maintenance des services. Les agents mobiles pourront éventuellement être disponibles pour "location".

1.1 Définitions et concepts de base

La mobilité du code peut être définie de façon non formelle comme la capacité de changer dynamiquement le lieu d'exécution, ou la source de provenance de fragments de

code durant l'exécution d'une application. Le code peut être arrêté, déplacé puis redémarré à une machine distante (i.e. les agents mobiles), ou une application peut utiliser du code situé sur une autre machine qui est transféré à l'exécution (code sur demande : applets). La technologie (code mobile) devrait permettre la création de nombreux services innovateurs. La triste réalité demeure que les applets sont actuellement la seule implémentation de code mobile déployée à grande échelle.

La convergence de la téléphonie et de la réseautique est illustrée par la *téléphonie Internet*. Les services avancés de téléphonie par l'Internet et le commerce électronique sont deux applications existantes qui devraient bénéficier des agents mobiles. La téléphonie Internet est le transport en temps réel de la voix et des applications multimédia sur l'Internet. Les services avancés de téléphonie sont ceux qui ne traitent pas du contrôle (établissement) des appels. Ces services peuvent être des services téléphoniques purs tels que la diversion des appels ou des services hybrides qui interfacent par exemple avec le courrier électronique (un courriel est envoyé pour avertir des appels manqués). Les services avancés peuvent être implémentés ou transportés par des agents mobiles, facilitant l'ajout et le retrait desdits services, en découplant totalement la gestion des services du contrôle des appels. Pour se servir des agents mobiles, tout noeud du réseau ou toute machine où l'agent pourrait se déplacer doit avoir un environnement d'exécution (plate-forme) d'agents mobiles. Ceci est similaire à l'architecture des *réseaux actifs*.

Les *réseaux actifs* (Wetherall, 1999) permettent l'injection de paquets appelés capsules qui contiennent des programmes dans le réseau. Le format d'une capsule est une extension du format d'un paquet IP. Une capsule ressemble à un agent mobile en ceci qu'elle se déplace à travers les nœuds actifs en respectant sa propre politique de migration. Si la capsule passe par un commutateur ordinaire, il fait suivre le paquet tout simplement selon la valeur de l'entête IP du paquet. Un noeud actif est un commutateur programmable (logiciel) qui implémente une interface que les capsules manipulent pour s'informer sur l'environnement et profiter des services disponibles dans le réseau. L'interface est unique pour tous les nœuds actifs du réseau. Chaque fois qu'une capsule invoque une méthode de l'interface, la méthode est traitée par le service indiqué dans le

champ d'entête de la capsule. Les seuls services offerts par les implémentations actuelles sont ceux de migration des capsules (multicast, point à point, etc.). Les réseaux actifs standardisent le traitement à chaque noeud (quelles méthodes invoquer) au lieu de standardiser le traitement effectué sur chaque paquet.

L'alternative au code mobile est le modèle client-serveur avec ses appels de procédure à distance (RPC), implémenté par les technologies comme CORBA ou Microsoft COM. Dans un programme, les appels de procédure à distance (APD) sont identiques aux appels à des procédures locales. Cette similarité est toutefois superficielle. Pour implémenter les APD, un "stub" est généré au client et au serveur. Les APD transitent par ces "stubs" qui ouvrent une connexion entre les hôtes clients et serveurs et transmettent la procédure invoquée et ses paramètres dans un message. Les connexions "pipe" permettent de retourner les résultats intermédiaires ainsi que de grandes quantités de données. Les "stubs" doivent être regénérés chaque fois que le programme change. Le support système est limité au formatage et à la transmission des données.

Le support système des agents mobiles ou des réseaux actifs inclue souvent un interpréteur, une couche qui traduit le code exécutable (généré selon un format unique) et invoque les primitives appropriées du système d'exploitation. Dans les langages non-interprétés, l'exécutable d'un programme varie selon le système d'exploitation, principalement parce que les programmes font des appels directs aux primitives desdits systèmes. Ce sont les interpréteurs qui permettent le code mobile.

1.2 Eléments de la problématique

Les services avancés de téléphonie Internet doivent être implémentés suivant une architecture qui sépare totalement la gestion des services de l'établissement des appels. Cette division permet à l'architecture d'offrir, en plus des services avancés de téléphonie, des services hybrides et des services non reliés à la téléphonie. Glitho (2001) évalue l'alternative qui consiste à modifier les réseaux intelligents (Intelligent Networks) pour insérer la gestion des services dans le réseau. Il démontre que cette option est limitée et, tout comme les deux standards utilisés en ce moment (H.323 et SIP), ne remplit pas les

critères exigés pour que les services en téléphonie Internet soient compétitifs. H.323 de ITU-T (1999) est un standard qui rappelle la téléphonie à commutation de circuit. L'autre standard (Handley *et al.*, 1999) est le Session Initiation Protocol (SIP) de l'Internet Engineering Task Force (IETF).

Dans la vie courante, les utilisateurs peuvent personnaliser les services Internet (comme le courriel) auxquels ils accèdent à partir d'un ordinateur personnel. Il leur est facile d'ajouter des services à leur ligne résidentielle et la dérégulation leur permet maintenant de changer de fournisseur avec aisance. La téléphonie Internet doit relever ces défis si elle veut être compétitive. Glitho (2000) indexe les critères que l'architecture pour les services avancés en téléphonie doit remplir ;il évalue H.323 et SIP par rapport à ces critères. Nous estimons que ces critères sont raisonnables et les reprenons à notre compte :

- le support doit être pour une large palette de services ;
- la création et le déploiement des services doivent pouvoir se faire rapidement ;
- l'usager doit avoir la possibilité de personnaliser ses services ;
- l'évolution du réseau (IN) et de l'infrastructure des services doit être indépendante ;
- de nombreux et différents fournisseurs de services doivent pouvoir entrer facilement sur le marché ;
- la gestion (insertion, retrait, mise à jour) des services doit être simple ;
- l'accès aux services doit être universel, i.e. indépendant de la position de l'usager et du moniteur qu'il utilise ;
- l'architecture doit permettre la collaboration avec les autres services déjà existants.

Les standards H.323 et SIP ont des résultats piteux par rapport à ces requis tels que le démontre Glitho (2000). Modifier les réseaux intelligents n'est pas plus porteur selon Glitho (2001). La solution pourrait donc être d'implémenter les services de téléphonie comme des capsules et développer des nœuds actifs pour avoir un réseau actif. Plus simplement, les services peuvent aussi être transportés dans des agents mobiles. Bien que les réseaux actifs permettent l'introduction rapide de nouveaux services, ils sont limités aux services de téléphonie et même pour ces cas là ne sont appropriés que pour les

applications de routage "multicast", notification de congestion et surveillance du réseau. Les réseaux actifs n'offrent pas l'accès universel et ne permettent pas de personnalisation. En bref, ils ne sont envisageables que si on est uniquement concerné pas l'insertion et le retrait rapide de services reliés à la gestion du réseau et qui n'interagissent pas avec les usagers.

L'accès universel et la personnalisation (préservation des données) sont deux motifs pour lesquels les agents mobiles sont préférables. Toutefois, l'informatique mobile propose des solutions à ces requis là. Pourquoi ne pas limiter les agents mobiles aux services à l'intérieur du réseau et utiliser les systèmes d'informatique mobile pour l'approvisionnement en service des moniteurs d'information portables ? Les moniteurs disposent en général d'une faible bande passante et sont même déconnectés du réseau s'ils sortent de la zone de couverture.

Les paradigmes de l'informatique mobile sont pour la plupart basés sur le modèle client-serveur avec ses APD. Ces paradigmes peuvent être classifiés en trois catégories (Jing *et. al.*, 1999): adaptation mobile, modèle client-serveur étendu et accès des données par des clients mobiles. L'adaptation mobile permet l'allocation dynamique de ressources aux applications et systèmes en fonction des changements dûs à la mobilité. En général, les applications sont obligées de sacrifier des fonctionnalités et de la performance. Le fonctionnement en mode déconnecté est compliqué et limité. Le système Rover en est le plus connu (Joseph *et al.*, 1997).

Le modèle client-serveur étendu requiert que les applications dans le serveur soient partitionnées (découpées) et optimisées pour la mobilité. Ceci permet de transférer certaines fonctionnalités du serveur au client ou vice-versa avant la déconnexion. L'approche nécessite le support d'un système tel que : InfoPad, CITRIX (voir site CITRIX) ou CODA pour son fonctionnement. Ici encore, les applications possibles sont limitées. Les paradigmes pour l'accès au données par des clients mobiles reposent sur la réplication des données contenues au serveur et la consistance des données en cache du client. Ils ne traitent pas la survivabilité des applications mobiles ou leur fonctionnement. Cette brève analyse montre les faiblesses intrinsèques des paradigmes de l'informatique

mobile qui les disqualifient pour notre architecture. Les agents mobiles se révèlent être la meilleure et unique approche qui permette de remplir tous les critères énoncés plus haut.

1.3 Objectifs de la recherche

Ce mémoire a pour objectif principal la spécification et l'évaluation du cycle de vie d'une architecture d'agents mobiles qui satisfait toutes les exigences énoncées dans les éléments de la problématique. Les services pourront être personnalisés à travers l'agent mobile et seront accessibles en tout temps et sur tout moniteur (l'agent peut résider sur le moniteur et se déplacer sur un autre au besoin). Plus spécifiquement, ce mémoire vise :

- la spécification détaillée et l'implémentation de l'architecture qui précisera le mécanisme d'approvisionnement en service, l'intelligence des agents (en opérations possibles), ainsi que les mécanismes de communication employés ;
- l'évaluation de performance de l'architecture ;
- la description, l'implémentation et l'évaluation de mécanismes de mise à jour de l'architecture et de ses services.

La mise à jour dynamique doit être traitée et une solution performante et efficiente sera proposée, car elle est primordiale pour les services de téléphonie. En effet, les services de téléphonie ne peuvent être arrêtés, ils doivent rouler continuellement et être mis à jour alors qu'ils sont en cours d'exécution.

1.4 Plan du mémoire

Ce mémoire comprend six chapitres. Au chapitre 2, nous présentons une revue, une évaluation critique et les applications des agents mobiles, des agents intelligents, des systèmes multi-agents et des systèmes de mise à jour dynamique. Ensuite, au chapitre 3, nous abordons l'implémentation de notre architecture. Au chapitre 4, nous expliquons les mécanismes de mise à jour des agents et des services. Au chapitre 5, nous procédons à une évaluation de performance de l'architecture, des mécanismes de mise à jour et à une évaluation d'intégrité de nos mises à jour.

CHAPITRE II

CONCEPTS D'AGENT ET APPLICATIONS

On distingue deux communautés qui œuvrent dans le domaine des agents : celle des agents intelligents et celle des agents mobiles. Les agents intelligents sont essentiellement des systèmes à base de connaissances ou des systèmes experts. De nombreuses applications commerciales basées sur ces systèmes sont actuellement disponibles dans les services d'information, de recherche d'emploi, de divertissement, d'assistance personnalisée sur l'Internet et de commerce électronique. Les agents mobiles, quant à eux, sont des agents qui se déplacent dans le réseau durant leur existence. Dans ce chapitre, nous passons en revue les concepts et applications des technologies agents. Dans un premier temps, nous examinons la technologie des agents intelligents et des systèmes multi-agents (SMA). Ensuite, nous retraçons l'évolution technologique des agents mobiles dont nous évaluons les avantages et les actuelles limitations. Ensuite, nous décrivons quelques-unes des applications d'agents mobiles développées jusqu'à présent. Enfin, nous traitons des mises à jour dynamiques et des techniques qu'elles utilisent.

2.1 Agents intelligents et systèmes multi-agents

Les agents, particulièrement les agents intelligents, sont l'objet de recherche depuis le début des années 80, principalement par les chercheurs en intelligence artificielle distribuée. Récemment, le Web a relancé l'intérêt pour le paradigme et d'importantes ressources sont investies dans la recherche, le développement et le déploiement des agents. Dans cette section, nous présentons les principes de base des agents intelligents. Nous abordons ensuite les architectures des systèmes d'agents. Pour finir, nous nous attardons sur plusieurs applications commerciales des agents intelligents.

2.1.1 Définitions et principes de base

Un *système multi-agent* est un ensemble d'agents qui coopèrent intelligemment pour réaliser une tâche. L'intelligence est distribuée entre les agents du système. Les systèmes à base de connaissances (SBC) sont des programmes qui infèrent des décisions étant donnés certains faits, en utilisant un ensemble de règles. Un SBC est un système expert quand le programme est conçu pour agir comme un expert dans un domaine précis.

Un *système expert* est principalement composé d'une base de connaissances et d'un moteur d'inférences qui implante un ou plusieurs mécanismes d'inférence. Ces systèmes sont aussi dotés d'un utilitaire qui explique comment une décision a été obtenue ou pourquoi une donnée est nécessaire. L'intelligence est couramment représentée par la logique des prédicats, des règles de production ou des réseaux sémantiques, etc. Les mécanismes d'inférence sont des stratégies de contrôle ou des techniques de recherche qui parcourent la base de connaissances pour arriver à des décisions (Rajeev *et al.*, 1996). Certains systèmes à base de connaissances sont capables d'apprentissage. La plupart d'entre eux apprennent par induction, d'autres par déduction.

Une *ontologie* est une spécification explicite d'une conceptualisation. Pour illustrer, considérons deux politiciens qui parlent d'un «système », puis deux ingénieurs logiciels qui parlent d'un «système ». Les deux groupes parlent de «système », toutefois, le «système » des ingénieurs est différent du système des politiciens. Dans ce cas-ci, les politiciens partagent la même ontologie, tout comme les ingénieurs. Seulement, les deux groupes utilisent des ontologies différentes. Pour simplifier, on peut dire qu'une ontologie est un langage commun et un vocabulaire qui sert de cadre de référence pour interpréter les messages échangés.

2.1.2 Architecture et ontologies

La recherche d'une architecture adéquate pour un SMA a mené à plusieurs spécifications différentes. La plus connue est celle de Shoham (1993). Les propriétés désirables d'un système multi-agents peuvent s'énoncer comme suit :

- offrir des mécanismes pour ajouter et retirer des agents de la société d'agents ;
- permettre la construction de société d'agents, c'est à dire de groupes d'agents qui collaborent pour atteindre un objectif précis. Les agents ne sont pas supposés avoir une connaissance au préalable des uns et des autres ;
- offrir des mécanismes pour la résolution de conflits.

A un degré moindre, un SMA devrait prévoir des mécanismes pour intégrer des agents hétérogènes (qui ne sont pas supposés collaborer) et réutilisables. Cette propriété est en fait primordiale pour les applications Internet. Toutefois, pour des solutions propriétaires, il est risqué d'interagir avec des agents inconnus.

L'architecture d'un SMA peut être décomposé en trois couches : typologie et structure, communication, coopération (Pelletier *et al.*, 2000). La première couche définit les agents et analyse leurs possibles activités. La couche communication détermine l'échange d'information entre les agents. La couche coopération établit le modèle de négociation (commerce électronique) entre agents.

Une ontologie commune permet aux agents intelligents de partager et de réutiliser leurs connaissances. Le terme ontologie est emprunté de la philosophie où il désigne une occurrence systématique d'existence. Nous utilisons des ontologies courantes pour décrire le langage d'un groupe d'agents de sorte qu'ils puissent communiquer sans nécessairement opérer selon une théorie commune. On dit qu'un agent souscrit à une ontologie quand ses actions observables sont consistantes avec les définitions de cette ontologie (Tom Gruber, 1994).

Pour communiquer, les agents ont besoin d'un langage commun. Les standards KQML (Knowledge Query and Manipulation Language) et ACL (Agent Communication Language) sont les langages de communication inter-agents les plus connus. Leur spécification est semblable à un protocole de passage de messages.

Les agents qui partagent la même ontologie peuvent utiliser KQML pour communiquer. Les deux langages sont basés sur des performatives où l'intention du message sur le receveur est spécifiée dans le message (dire, demander, etc.).

Dans le commerce électronique, XML-EDI est utilisé pour les transactions électroniques entre agents. EDI est une extension de XML qui spécifie le contenu des documents d'affaires pour les transactions Web (Glushko *et al.*, 1999 ; site disa.org). KQML et XML ajoutent de la flexibilité et de l'adaptabilité aux systèmes d'agent, ce qui leur permet d'effectuer du "shopping" et de réaliser des transactions sur le Web.

2.1.3 Applications des agents intelligents

Les agents intelligents sont utilisés dans les applications d'entreprise virtuelle, de réseautique scientifique (qui permettront de traiter les ordinateurs en réseau comme un unique méta-ordinateur), de manufacture intelligente et de gestion de distribution de l'énergie. Ils sont déjà commercialement utilisés dans des services et applications de nouvelles et d'information, de shopping, de ventes aux enchères, de navigation personnalisée et de recherche d'information.

Nouvelles et informations

Les premières applications d'agent disponibles commercialement concernaient les agents moniteurs. Ces systèmes parcouraient les données et avertissaient l'usager quand un événement d'importance se produisait. *E-Watch* (Site ewatch.com), *ZDNet* (Site zdnet.com) et *Excite* (Site excite.com) offrent ce service de nouvelles et d'information. Les agents de ce type sont nombreux sur les sites boursiers et les sites de commerce électronique.

Shopping

Frictionless (Site frictionless.com) permet aux usagers de comparer les prix et les caractéristiques des produits lorsqu'ils font des emplettes par l'Internet. L'usager peut remplir une fiche décrivant son profil, puis choisir un produit et ses caractéristiques. Son agent demandera alors ce produit aux différents marchands. Les produits satisfaisants sont ensuite présentés à l'usager et ordonnés selon leur correspondance au profil de l'usager, leur prix, les politiques du vendeur (livraison, retour, échange, etc.). D'autres sites de shopping (jango.com) utilisent des agents.

Enchères

AuctionBot (Site auction.eecs.umich.edu) est un serveur Internet qui permet de tenir des enchères sur n'importe quel produit. Il est situé à l'université de Michigan. Une interface sur le site permet aux usagers de créer et de spécifier les caractéristiques de leur agent. Cet agent peut alors prendre part aux enchères sur le site. Les usagers créent une nouvelle enchère en spécifiant le type de l'enchère, un prix de départ et une méthode de résolution en cas d'égalité. Plusieurs sites d'enchères traitant de produits spécialisés sont disponibles sur Internet (Voir Sites Enchères).

Navigation Personnalisée

Alexa (Site alexa.com) est un utilitaire gratuit qui est compatible avec n'importe quel fureteur et qui fournit de l'information à propos de chaque site visité. Il affiche les statistiques de chaque page visitée dans sa propre fenêtre. Il est aussi utile pour le commerce électronique car il vérifie l'information sur les propriétaires de chaque site avant que l'usager n'utilise sa carte de crédit. Le produit WBI (Voir Site almaden.ibm.com/cs/wbi) de IBM, qui est un proxy programmable de Web serveur, est un autre exemple de cette catégorie.

Recherche d'information

La technologie "Push" et les "Bots" (ro"Bots" Internet) sont deux champs de la recherche d'information qui utilisent des agents. Les moteurs de recherche sur le Web envoient des "Bots" qui parcourent les serveurs et collectionnent des listes d'URLs. En bref, les "Bots" fouillent au travers des données. Ils ont un grand potentiel pour la dissémination d'information, car ils peuvent détecter des patrons dans de grandes quantités de données.

Les "Bots" sont des agents statiques qui ont accès seulement aux données publiques disponibles par l'intermédiaire des serveurs Web. Les problèmes rencontrés par les agents mobiles proviennent de ces différences. Un agent mobile effectuera des transactions locales qui ralentiront le serveur et devra éventuellement accéder à des données privées, ce qui pose des problèmes de sécurité. Les "Bots" ne coopèrent pas entre eux et ne transportent pas leur état quand il se déplace.

La technologie "Push" est en fait un ensemble de technologies utilisées pour envoyer de l'information à un usager sans qu'il n'en fasse la demande (Site jm.acs.virginia.edu). Les médias publics tel la radio et la télévision sont basés sur cette technologie. Le Web est basé sur du "Pull" (l'usager doit cliquer). La technologie "Push" sur le Web permet à l'utilisateur de choisir l'information qui lui sera envoyée, le moment et le format de l'envoi.

La technologie "Push" permet aux entreprises d'envoyer de l'information à des audiences ciblées. Elle permet notamment de naviguer sur le Web sans utiliser de fureteur et facilite la distribution des nouvelles versions des logiciels. Les agents conviennent naturellement à cette technologie puisqu'un agent peut être envoyé sur un serveur et filtré l'information localement. Seule l'information pertinente est retournée à l'expéditeur.

Le réseau Poincast (Site www."Pointcast".com) est le produit le plus avancé de la technologie "Push" qui utilise des agents. Il permet aux usagers de personnaliser leurs nouvelles sur le sport, la météo et les valeurs boursières. C'est un utilitaire gratuit qui dispose d'une interface graphique et qui délivre les nouvelles sur l'ordinateur personnel de l'usager.

2.2 Agents mobiles

Les agents mobiles empruntent beaucoup de la migration de processus qui fait référence au transfert de processus entre deux ordinateurs. Un processus est une abstraction des systèmes d'exploitation qui comprend le code, les données et l'état d'exécution d'une application. Initialement, la migration de processus était typiquement implantée au niveau du système d'exploitation. Le principal défi était de transférer l'état interne du processus contenu dans le noyau du système d'exploitation. Il était aussi difficile de transférer les ressources systèmes, i.e. les pointeurs de fichiers, les pilotes d'interface et de périphériques, etc. C'est pourquoi la migration de processus a d'abord été implanté avec les systèmes basés sur les messages où l'interaction du processus avec le monde se fait au travers de canaux précis (Powell *et al.*, 1998). La migration de processus basée sur les appels au noyau du système d'exploitation a suivi (Douglis *et al.*, 1998). Ces systèmes ont connu un certain succès, même si aucun d'eux n'a pu s'imposer en environnement réel, à cause de leur complexité.

La migration de processus a introduit les notions de *code mobile* et d'*objet mobile* (un graphe de l'objet est créé et transféré). Les agents mobiles ont enrichi ces notions et sont maintenant implantés en langage interprété qui supporte du code mobile. Ils sont indépendants des systèmes d'exploitation.

Selon leurs promoteurs (White, 1998), les agents mobiles permettent de dépasser les limites du modèle client-serveur. Cependant, jusqu'à présent, les expériences ne sont pas parvenues à démontrer une réduction significative du trafic réseau, sauf pour certains cas particuliers d'application à petite échelle. Les mécanismes pour augmenter la tolérance aux fautes des agents demeurent inadéquats sur l'ensemble des systèmes; la tâche repose donc sur les épaules du programmeur. De pertinentes réserves sont soulevées sur la sécurité des agents, leur contrôle et leur communication.

2.3 Avantages et limitations

Dans cette section, nous présentons les principaux avantages et limitations des agents mobiles. Cela nous permettra de mettre en évidence les défis qui sont à relever pour populariser leur utilisation.

2.3.1 Avantages

L'un des avantages souvent évoqués est que les agents mobiles permettent de dépasser les limitations du modèle client-serveur. En effet, les limitations de la machine cliente telles que la puissance des processeurs, son débit, la taille de son espace mémoire peuvent être atténuées si l'agent s'exécute près des données. Par exemple, envoyer un agent vers une base de données qui doit être examinée selon un algorithme particulier peut augmenter la performance, comparé au cas où les requêtes seraient envoyées à partir d'une machine distante. Toutefois, les résultats d'expériences (Johansen *et al.*, 1999) contredisent ces affirmations. En effet, on observe des gains de performance dans certaines circonstances particulières (Ranganathan *et al.*, 1998; Outtagarts *et al.*, 1999). La plupart du temps, il n'y a pas de gain ou alors ils sont fractionnaires. Les meilleurs résultats sont obtenus avec une combinaison de client-serveur et de code mobile. Les gains dépendent toujours du réseau : plus son débit est faible, plus grands seront les gains.

Un autre avantage proclamé des agents mobiles est qu'ils permettent une interaction asynchrone. En effet, les agents mobiles implantent la programmation à distance au lieu des appels de fonction à distance. L'ordinateur doit donc être connecté assez longtemps pour déplacer l'agent sur le réseau et plus tard l'accueillir. Ceci est particulièrement utile pour les "palmtops" et ordinateurs mobiles qui ne sont généralement connectés que de façon intermittente. De plus, on peut s'assurer que les agents survivent aux fautes matérielles. Si une machine tombe en panne, l'agent peut être enregistré sur le disque dur de la machine ou il peut se déplacer vers un autre hôte pour revenir plus tard. Une fois que la machine est remise en marche, l'agent continue sa tâche et retourne vers l'hôte qui l'a envoyé quand il a obtenu des résultats. Une

interaction continue ne nécessite donc pas de communication continuelle. Cette propriété permet de concevoir des applications d'intégration des médias de communication. Une telle application est dite *messagerie sans interruption* où l'agent peut chercher à joindre l'abonné sur plusieurs terminaux, de sorte qu'il peut lui livrer un message urgent, peu importe le média d'origine. Cette commutation de média est grandement compliquée avec le modèle client-serveur.

Presque toutes les plates-formes d'agents mobiles actuelles permettent une interaction asynchrone. Cependant, les agents qu'on peut développer avec leurs outils ont une autonomie et une intelligence limitées. Par conséquent, ils n'exploitent pas les possibilités d'interaction asynchrone.

Par ailleurs, les agents seraient plus faciles à personnaliser et ajouteraient de la flexibilité aux systèmes. En effet, un agent mobile peut être développé comme un «bean » (Java Beans). La technologie « Java Beans » permet l'interopérabilité entre composants provenant d'applications différentes. Un « bean » est développé et testé indépendamment; par conséquent il est plus facilement réutilisé. En effet, le code source qui définit un composant « Java Beans », ne fait référence à aucun autre composant. Les références sont établies à l'exécution. Ces références peuvent être établies par le contenant qui tient le composant « Java Beans » ou par le composant lui-même. Cette propriété permet de raccourcir le temps de développement d'une application utilisant des « Java Beans », donc éventuellement d'une application basée sur des agents. Les « Java Beans » offrent les mêmes avantages que ActiveX et VBX pour les applications d'interface usager.

Une *place* est un environnement d'exécution pour agents. Typiquement, c'est une plate-forme pour agents en exécution sur une machine. Les agents se rencontrent dans des places. Un serveur peut donc être configuré comme une place. Ainsi, quand une nouvelle application doit être installée sur le serveur, un agent mobile qui implémente cette application est envoyé au serveur. L'agent s'agrège donc au serveur et le serveur offre cette nouvelle application. Les applications simples peuvent donc être rajoutées facilement, i.e. chaque usager peut rajouter les applications qui l'intéressent.

Comparativement, les applications statiques qui utilisent des appels de procédure à distance impose que le serveur soit arrêté, puis mis à jour. De plus, comme la nouvelle interface sera disponible à tout le monde, pour un serveur public ceci implique qu'il faut une décision d'affaires avant d'ajouter une nouvelle application, peu importe sa simplicité ou le nombre de ses utilisateurs.

Les implications de ce concept de serveurs extensibles sont impressionnantes :

- les applications sur les communicateurs personnels (""palmtop"", ordinateur mobile) pourront être configurées comme une collection d'agents;
- les réseaux et serveurs publics deviennent des plates-formes pour agents mobiles (James, 1998).

Ces concepts sont déjà utilisés avec les plugiciels (plug-ins) des fureteurs (Netscape, Internet Explorer) et constituent des avantages bien réels. Ainsi, plusieurs usagers accédant à un même serveur peuvent en avoir des vues totalement différentes, car chacun utilisera des agents personnalisés offrant des services différents. La question qui se pose alors est de savoir comment maintenir et coordonner un tel système.

Un autre des supposés avantages des agents mobiles est qu'ils sont plus faciles à programmer et augmenter. Cela provient de l'observation que les agents offrent souvent une meilleure représentation du monde réel. Par exemple, pour les utilisateurs d'ordinateur qui sont mobiles, les travaux qu'ils commencent au bureau doivent souvent être poursuivis même s'ils quittent leur ordinateur. Une façon de faire serait de déléguer ces tâches à un ordinateur qui continuera le travail pendant que l'usager sera absent.

En attendant que les difficultés avec les ontologies soient surmontées, les agents pourraient implémenter des technologies comme Jini (Site www.sun.com/jini) qui permet à des applications différentes provenant de différents vendeurs de s'entre-identifier et de s'entre-utiliser. Notons enfin que les mécanismes de communication utilisés actuellement sur toutes les plates-formes d'agents mobiles sont précaires.

2.3.2 Limitations des agents mobiles

Les agents mobiles seraient dangereux à utiliser. En toute rigueur, accepter des agents mobiles n'est pas nécessairement différent d'accepter du code mobile comme on le fait avec les applets de Java. Le risque peut être comparé à celui lié au fait d'accepter des courriels avec des entités actives, tels les documents Word qui contiennent des macros (Milojičič *et al.*,1998). Si les agents sont restreints à communiquer au travers d'interfaces bien définies et sont limités dans leurs actions (comme le modèle sandbox de Java le permet), alors les risques sont considérablement réduits.

Malheureusement, les interfaces sont peu ou pas définies sur la plupart des plates-formes. De plus, les problèmes de sécurité ne sont pas limités à un usage abusif de l'hôte par l'agent, ils incluent aussi l'usage abusif de l'agent par l'hôte ou par d'autres agents. Greenberg *et al.* (1998) ont cité les attaques les plus probables sur un agent ou un hôte : refus de service, accès aux données personnelles, harcèlement, ou une combinaison de celles-ci. La plus dangereuse est celle de l'accès aux données personnelles ou un agent peut utiliser un canal pour retransmettre des données tout en respectant les interfaces qui lui sont imposées. Les attaques qui sont une combinaison des techniques sont les plus difficiles à retracer.

Les solutions de sécurité envisagées jusqu'à présent pour les agents mobiles sont inadéquates parce qu'elles sont calquées sur celles qui étaient employées sur les vieux systèmes qui n'acceptaient pas de programmes extérieurs. Des mesures de protection adéquates n'ont pas encore été conçues pour les systèmes ouverts (Internet). Ainsi, la sécurité des agents est un domaine en pleine évolution où beaucoup reste à faire. N'empêche, des solutions sont disponibles pour la plupart des problèmes, excepté l'usage abusif de l'agent par l'hôte.

Faire communiquer des agents nécessite le recours à des mécanismes d'interopérabilité de haut niveau entre des programmes (Finn *et al.*, 1998). Ceci est difficile à réaliser étant donné que :

- différents langages sont utilisés pour programmer les agents ;
- les plates-formes et les systèmes d'exploitation sont souvent différents ;

- peu d'hypothèses peuvent être faites sur l'état interne des agents.

Puisque les langages KQML et ACL ne spécifient pas la syntaxe ni le contenu sémantique des messages, deux langages généraux pour spécifier le contenu des messages ont été créés. Il s'agit de KIF (Knowledge Interchange Format) et FIPA SL. Malgré cela, d'autres questions se posent :

- comment traduire d'un langage de programmation à un autre ?
- comment le sens des concepts et des relations peut-il être préservé entre plates-formes ?
- Comment partager la connaissance ?

Les agents sont actuellement limités parce que les ontologies ne permettent pas un partage de connaissances. Pire, sur la plupart des plates-formes, les mécanismes internes de communication inter-agents sont inadéquats. Ceci devrait s'améliorer au fur et à mesure que des applications d'agent seront développées. De plus, une plate-forme d'agent mobile pourrait s'établir comme choix consensuel et ainsi annuler certains des problèmes actuels.

Bien que plusieurs systèmes pour agents mobiles aient été développés, aucun d'entre eux n'a réussi à populariser les applications d'agents mobiles. Le nombre de serveurs capables d'accepter des agents mobiles est réduit, et il n'existe pas de systèmes ou d'applications répandues qui acceptent des agents. Si l'on fournit des plugiciels pour des environnements répandus comme les fureteurs Web, ce problème de serveur pourrait être résolu. Enfin, il convient de mentionner qu'il n'existe pas de standard relatif au nom, à la localisation et au contrôle des agents, encore moins de directive ou de normes sur la façon de gérer un serveur qui accepte des agents, la variété de l'activité à ce serveur étant beaucoup plus grande que celle d'un serveur Web.

2.4 Systèmes d'agents mobiles

On distingue trois approches pour concevoir et implanter une plate-forme d'agents mobiles (Karmouch *et al.*,1998). La première consiste à recourir à un langage de programmation qui comprend des instructions pour les agents mobiles. Compaq[MD] a

essayé sans succès cette approche avec le projet *Obliq* (Voir Site research.compaq.com). La deuxième approche consiste à implanter le système d'agents mobiles comme des extensions du système d'exploitation (Site cs.uit.no). Enfin, la dernière approche construit la plate-forme comme une application spécialisée qui tourne au-dessus d'un système d'exploitation. La plupart des systèmes utilisent cette approche qui résulte souvent en une collection de librairies Java (*Voyager*, Aglet, Concordia, Mole, Odyssey). Sinon, ils sont écrits en langage de scripts avec un interpréteur et des utilitaires d'exécution pour leur utilisation (D'Agent, Ara).

Tous ces systèmes ont une architecture de serveur. Plusieurs d'entre eux utilisent l'approche «sandbox » où les permissions de l'agent mobile sont limitées et contrôlées. Ces systèmes classifient les agents en deux catégories : les agents sûrs et les agents non sûrs. Nous présentons ici quatre plates-formes représentatives des systèmes pour agents mobiles : Aglets, Mole, Sumatra et *Voyager*.

Aglets

Aglets est une plate-forme d'agents mobiles developée par IBM Japon (Site trl.ibm.com/aglets). Un Aglet est un objet Java mobile qui visite des environnements d'exécution (serveur) Aglets. L'architecture des Aglets est similaire à celle des applets Java. Le système Aglets a son propre protocole pour le transfert des Aglets entre hôtes : Aglet Transfer Protocol.

Mole

Mole est une plate-forme d'agents mobiles construite à l'université de Sttutgart en Allemagne. Elle implémente la migration partielle, où seuls les données et l'état de l'agent sont transférés. La migration partielle a été développée après que les bâtisseurs du système se soient rendus compte que la migration totale (déplacement du processus d'exécution de l'agent) était trop coûteuse quand il s'agissait d'agents « multi-threads » (Baumann *et al.*, 1998). Un agent est traité comme une grappe d'objets Java, un ensemble fermé sans aucune référence avec l'extérieur, excepté avec le système hôte.

Sumatra

Sumatra a été développé pour mesurer la performance des agents dans la gestion des réseaux. Il implante une application *Komodo* qui surveille l'état (les délais) du réseau. L'application test est *Adaptalk*, une application de «chat» Internet. Les textes entrés par les usagers sont acheminés aux destinataires par des agents mobiles qui déterminent dynamiquement leur trajectoire en tenant compte des délais du réseau. Des améliorations de performance significatives ont été observées dans certains cas. Dans ce contexte, une place est appelée un interprêteur exécutant sur une machine, et un agent est un groupe d'objets (Ranganathan *et al.*, 1998).

Voyager

Voyager (Site www.objectspace.com) est un *Object Request Broker* (ORB) écrit en Java qui offre des services pour agents. Il intègre en conséquence les technologies ORB, CORBA, RMI. De plus, *Voyager* supporte les « Java Beans ». Ainsi, tout objet peut être traité comme un agent pourvu qu'il soit sérialisable. Sérialiser un objet consiste à créer récursivement le graphe de l'objet et des objets qu'il référence avant de les transférer entre machines.

Voyager permet d'envoyer des messages asynchrones (sans réponse) et synchrones (comme pour les appels de fonctions à distance). On peut aussi envoyer des messages qui seront livrés à une date ultérieure sous certaines conditions. Les messages envoyés à un agent qui s'est déplacé sont redirigés à son nouvel emplacement. Si l'agent est en cours de déplacement, les messages sont bloqués jusqu'à ce qu'ils soient restitués à la destination. *Voyager* supporte aussi le « multicasting », où un message peut être envoyé à plusieurs hôtes en parallèle. Chaque agent doit alors s'enregistrer à un espace sur son hôte. Un message multicast est envoyé à tous les hôtes qui sont connectés entre eux.

Voyager contient un service de noms qui permet d'associer un nom à un objet. Il est possible de se connecter par la suite à cet objet en utilisant ce nom. Pour localiser un

objet, on utilise le service de nom et l'alias de l'objet. Le nom d'un objet est toujours composé de son URL suivi de son alias.

Pour déplacer un objet, il suffit d'invoquer la méthode *moveTo()* sur l'objet, en spécifiant la destination où l'objet doit se déplacer. Les références aux objets locaux deviennent des mandataires (proxys). Le Tableau 2.1 résume les caractéristiques des systèmes d'agents mobiles.

Tableau 2.1 Caractéristiques des systèmes pour agents mobiles

Système pour agents mobiles	Sécurité	Portabilité	Mobilité	Communication	Gestion de Ressources	Contrôle
Aglet	Sandbox	Java	Aglet Transfer Protocol	Evénement, message objet	Java	Oui
Mole	Java	Java	Modèle Java amélioré avec code server	Evénement	Java	Oui
Voyager	Sandbox, canaux protégés	Java	Java serialization, reflection	Evénements Distribués	Java	Oui
Sumatra	Java	Java	Java serialization	Signaux	Oui	Oui

2.5 Applications des agents mobiles

Bien que plusieurs applications centrées sur les agents mobiles aient été développées, peu sont allées au-delà de l'état expérimental et peu sont déployées commercialement. Les mesures collectées de ces applications sont non concluantes. Par contre, grâce à la flexibilité du paradigme agent, des applications pionnières peuvent être développées beaucoup plus facilement qu'avec le modèle client-serveur. Ces résultats couplés aux efforts de recherche actuels des universités et de l'industrie indiquent que plusieurs autres applications sont à venir. Dans cette section, nous passons en revue les applications des agents mobiles en télécommunications, au commerce électronique et à l'administration de système.

2.5.1 Télécommunications

Un Agent Communicateur Personnel (ACP) est un agent mobile qui doit livrer un message au destinataire, peu importe le média d'origine et le média de destination – avertisseur, téléphone, téléphone cellulaire, ordinateur personnel, ordinateur mobile. Par exemple, la seule façon de délivrer un courriel urgent à un usager pourrait être au travers d'un téléphone mobile. L'agent doit donc utiliser un convertisseur de texte à voix et délivrer un message vocal à l'usager (Abu-Hakima *et al.*, 1998).

L'Institut pour les Technologies de l'Information (IIT) du Conseil National de Recherche du Canada (CNRC) a développé un environnement de test pour évaluer deux applications d'intégration des médias (SPIN). Nous avons déjà présenté brièvement la première "Seamless Messaging" (SM) ; la seconde concerne la gestion intelligente des réseaux. Pour évaluer ces applications, ils ont mis en place un réseau local hétérogène. Ce réseau est composé de 30 ordinateurs personnels, un routeur SS7, un serveur CTI, une station de base mobile, une antenne mobile pour l'accès au réseau LAN, une passerelle ATM, une passerelle pour avertisseur, plusieurs téléphones (fixes, mobiles et cellulaires) et ordinateurs mobiles. Un agent diagnostic réside sur chacun des médias et permet de savoir si le média est en fonctionnement ou non.

Les agents sont appropriés pour le "Seamless Messaging", parce que ce sont des entités logicielles qui peuvent représenter l'usager, filtrer son information et, si quelque chose d'important se produit, ils peuvent utiliser le réseau pour contacter l'usager et l'avertir. Avoir un agent qui représente l'usager est un paradigme plus pratique que d'avoir un serveur centralisé qui fera plusieurs appels d'interfaces aux différents médias présents dans le réseau.

Le ACP de l'usager réside sur son ordinateur personnel. Il peut être configuré par téléphone ou en démarrant l'application de "Seamless Messaging" sur l'ordinateur. Il y a cinq agents dans l'application de SM. Le premier surveille les messages qui arrivent et les formate selon un standard. Le second est ACP ; il suit un ensemble de règles que l'usager spécifie quand il démarre l'application. Le troisième est l'agent secrétaire, il collabore avec l'ACP quand le message est «contactez-moi ». Sinon, il envoie

directement le message à l'agent chargé de la gestion du média concerné. L'agent fournisseur de service remplit des services spécialisés (conversion de texte à voix, de voix à texte). Quand l'usager ne peut être joint, l'agent secrétaire envoie le message à l'agent diagnostic qui l'enregistre dans une boite de messages universelle.

2.5.2 Commerce électronique

Tabican est une place de commerce électronique où se négocie l'achat et la vente de billets d'avion et de chambres d'hotel. Bâtie sur la plate-forme Aglet, elle a été conçue pour accueillir des milliers d'agents. Les ressources systèmes telles que les bases de données sont accédées exclusivement par le serveur *Tabican* qui fonctionne comme toutes les places de commerce électronique. Les vendeurs envoient leurs agents qui rencontrent les agents des consommateurs et discutent des prix. Les agents des promoteurs de voyage décrivent les voyages. Ceux des consommateurs contiennent l'ensemble des spécifications de leurs propriétaires. *Tabican* gère une base de données de types, appelée AMPM (Aglet Meeting Place Middleware) qui conserve l'information sur le type des messages échangés entre agents. Quand un agent arrive dans le système, il obtient un protocole d'interaction de la base de données AMPM. Il se sert ensuite de ce protocole pour communiquer avec les autres agents. Ceci permet à des agents non familiers développés indépendamment d'interagir. Le système contient plusieurs places d'échange. Les agents des consommateurs se promènent de place en place pour les meilleures réductions. Le système contient des agents de publicité qui vont de place en place signaler quand un nouveau produit est ajouté.

Mysimon.com [34] permet aux usagers de comparer les prix des marchands avant de faire un achat sur l'Internet. Il est considéré comme une application d'agents mobiles parce qu'il envoie plusieurs agents mobiles (qui effectuent un seul déplacement) qui font des recherches en parallèle et reviennent avec les meilleures opportunités sur le produit. Les agents peuvent être entraînés à rechercher selon les habitudes de l'utilisateur, à la manière d'un agent virtuel d'apprentissage (Virtual Learning Agent).

Guideware™ permet de développer des applications d'affaires, d'automatisation des ventes et de support à la clientèle. Il est cité comme une application d'agents mobiles dans plusieurs revues et sa page Web proclame qu'il utilise le paradigme d'agent mobile. Toutefois, la documentation technique sur son implantation est introuvable. Ceci peut être dû au fait que c'est une application pionnière dans le domaine des agents mobiles.

2.5.3 Administration de systèmes

Installer et maintenir des logiciels est ardu lorsque les machines sont géographiquement éloignées et que leur nombre croît. Vu qu'il est souvent nécessaire d'avoir une vue locale du système afin de pouvoir résoudre le problème, l'administrateur doit se déplacer sur de grandes distances (ville, province, pays, continent). Les agents mobiles simplifient cette tâche puisqu'ils peuvent se déplacer à ces nœuds et procéder à des mises à jour périodiques.

Jumping Beans[MD] de *Ad Astra Engineering* permet à un administrateur de système de gérer son réseau à distance. Leur produit permet de construire des agents mobiles qui installeront et maintiendront les logiciels à distance. L'administrateur doit simplement indiquer les logiciels à surveiller et à modifier.

Utiliser des agents mobiles pour la maintenance est principalement une question de convenance dans le cas où il s'agirait d'un Intranet. Toutefois, ceci est un avantage important pour un vendeur de service par l'Internet qui doit distribuer et maintenir sans interruption des services sur la machine de chaque usager pouvant être en assez grand nombre.

2.6 Systèmes de mise à jour dynamique

Effectuer une mise à jour dynamique consiste à modifier tout ou partie d'une application pendant que cette application est en cours d'exécution. Beaucoup de travail a été fait sur les mises à jour dynamiques. Nous révisons les contributions les plus récentes

et examinons leurs propriétés. Les systèmes logiciels de mises à jour dynamiques peuvent être classifiés en trois catégories : ceux qui demandent du support système, les langages de programmation qui ont des capacités de remplacement dynamique et les architectures logicielles qui permettent l'évolution dynamique du code. Les systèmes matériels de mise à jour dynamique sont basés sur des matériels redondants qui fonctionnent en parallèle durant un certain temps avant que le nouveau produit ne prenne la charge exclusive. Nous commençons par réviser *Erlang* (Armstrong *et al.*, 1996) puisque c'est à partir du travail effectué sur ce langage que nous avons dérivé notre solution Java.

2.6.1 Erlang

Erlang est un langage de programmation concurrent, symbolique et déclaratif développé aux laboratoires Ericsson et Ellemtel à Älvsjö en Suède. Son modèle de concurrence est similaire au langage de spécification et description pour le comportement des commutateurs de télécommunications nommé SDL (CCITT, 1999). Sa syntaxe ressemble à celle de ML non typé (Wikstrom, 1987). Des inter-compilateurs Erlang à ASN.1 (ITU-T 1997) sont disponibles, ainsi que des interfaces pour le système X-Windows. ASN.1 est un langage de description des types, standard pour décrire les formats des données utilisés pour la spécification des protocoles de communication.

Erlang est utilisé pour implémenter des commutateurs de communication de grandes tailles et les systèmes temps réel de contrôle tels que les systèmes de contrôle de trafic aérien. Il permet à un système d'opérer continuellement à travers des routines de chargement dynamiques de code qui permettent de changer du code dans une application en cours d'exécution. Il est aussi possible d'avoir plusieurs versions du même code qui exécutent concurremment dans l'application. Les programmeurs désignent explicitement quelles activités doivent être représentées dans des processus parallèles. Toutes les interactions entre processus se font par messages asynchrones puisque Erlang n'a pas de mémoire partagée. Il est apparent qu'*Erlang* implémente un mini système d'exploitation (un serveur de fichiers et une console sont disponibles).

Ces fonctionnalités de système d'exploitation limitent sa portabilité, le programmeur doit assumer le fardeau d'écrire des ports pour les autres langages de programmation. *Erlang* utilise des ports pour communiquer avec le monde extérieur.

De plus, les variables en Erlang ne peuvent être assignées qu'une seule fois, i.e. une variable ne peut être changée même quand une nouvelle version du code est dynamiquement chargée. Ces faiblesses couplées à la taille de distribution d'Erlang (12 Méga-Octets), joue contre son adoption comme plate-forme de choix dans le cadre d'une architecture pour la mobilité des services indépendante du système et adaptable aux petits moniteurs d'information ("palmtop", téléphone cellulaire, etc.).

2.6.2 Langages de programmations fonctionnels

Les langages de programmation fonctionnels modernes permettent des fonctions de haut-ordre. Une fonction de haut-ordre est une fonction qui peut être traitée comme toute valeur courante dans un programme. Elle peut donc être conservée dans des structures de données, passée en argument et retournée comme résultats. Les fonctions peuvent donc être appliquées à des portions de code différentes et être bâties à partir de la composition d'autres fonctions. ML (fortement typé, premier langage à offrir les fonctions polymorphiques), Lisp (Graham 1995) et Haskell sont trois exemples dans cette catégorie. Malheureusement, les langages fonctionnels sont encore restreints à une niche.

2.6.3 Système d'évolution dynamique bas niveaux

Par bas-niveaux, nous indiquons les systèmes qui demandent du support du système d'exploitation, de la machine virtuelle ou du compilateur. Le statut de la plupart des projets documentés est inconnu (Conic, Podus, Argus, etc.). La contribution active la plus significative est celle de Malabarba *et al.* (2000) qui ont développé une machine virtuelle Java qui permet des classes dynamiques. L'unité de changement est la classe. Les objets sont changés de façon assez transparente durant l'exécution. L'implantation inclut une infrastructure pour la sécurité.

Une première limitation est que le programmeur ne peut choisir quel objet de la classe doit être mis à jour. Conséquemment, l'état interne de tous les objets dynamiques est perdu après chaque modification. Les auteurs ont aussi choisi délibérément de permettre seulement une seule version active d'une classe par programme. Ce choix est surprenant puisque l'ajout de complexité due à la gestion de plusieurs versions est largement compensée par le fait que la coexistence assure la continuité du programme. Bien entendu, la solution souffre aussi du fait qu'elle n'est pas facilement portable.

2.6.4 Systèmes d'évolution dynamique de haut niveau

Hjalmtysson et Gray (1997) ont développé des classes C++ dynamiques. L'unité de changement ici aussi est la classe. Ils utilisent les templates C++ pour créer des proxies de classes dynamiques qui servent de liens avec les classes réelles. Les proxies dynamiques sont des classes abstraites pures. Pour permettre l'édition des liens à l'exécution, les méthodes sont appelées à travers une "jump" table qui est chargée au démarrage du système. Aucun support système n'est nécessaire et plusieurs versions de la même classe peuvent être actives au même moment.

Toutefois, les developeurs ne peuvent choisir quels objets mettre à jour et plusieurs versions de la même classe peuvent être actives seulement parce que les nouveaux objets refléterons toujours la dernière version de la classe. Les usagers ont le choix entre garder tous les anciens objets ou les détruire tous (restriction non nécessaire). La solution pêche aussi par le fait qu'elle détruit l'expressivité de l'héritage dans les langages de programmation par objets. Plus concrètement, les sous-classes d'une classe dynamique ne peuvent être utilisées là où la classe dynamique devrait l'être. Ceci annule une des raisons d'être de la programmation orientée objet. Une solution médiocre est de traiter toutes les classes dynamiques comme des classes finales. Ceci limite leur utilité. De plus, les auteurs admettent que le comportement des méthodes statiques est inconnu. Ceci est une inquiétude légitime puisque ce sont les auteurs qui ont développé les mécanismes d'édition de liens tardifs.

Oreizy *et al.* (1998) présente une architecture d'évolution dynamique où les programmes sont divisés en composants et communiquent à travers des connecteurs. Pour supporter les changements dynamiques, les connecteurs peuvent diverger toutes les communications destinées aux anciens composants vers les nouveaux. L'unité de changement le composant est généralement grande puisque chaque composant est obligé d'avoir une partie haute et une partie basse pour les communications (ports) dans les deux sens. L'architecture est prometteuse pour les applications distribuées car elle permet des changements dynamiques que ni CORBA, ni Microsoft COM n'offre.

Toutefois, la complexité des connecteurs est trop grande. En effet, les connecteurs doivent garder les messages en file lors d'un changement et être capable de défiler ces messages lorsque deux composants sont reconnectés. Si les connecteurs ont un type, alors changer le type du connecteur affecte tout le système. Par contre, s'ils n'ont pas de type on doit reconstituer leur représentation à partir d'une base abstraite sur laquelle tous les composants sont d'accord, augmentant ainsi les dépendances inter-composants. *Drastic* de Evans et Dickman (1999) utilise des mécanismes similaires

2.7 Synthèse des problèmes

Les applications mobiles complètent les autres techniques de programmation en ajoutant beaucoup de flexibilité aux systèmes. Elles simplifient des architectures comme celles des serveurs extensibles, où un agent peut représenter un service et être ajoutés au serveur sans qu'on ait besoin d'interrompre le serveur. Il est ainsi possible de contrôler la visibilité de ce service ; différents usagers peuvent donc se créer des ensembles quelconques de service sans interrelation.

Les autres arguments en faveur du paradigme agent n'ont pas encore pu être indiscutablement établis. Les mesures provenant des expérimentations sont non concluantes, à cause de différents facteurs : durée de l'expérience, taille de l'expérience, etc. Pour assombrir encore plus le tableau, les réseaux sont maintenant dotés de bandes passantes de plus en plus grandes ; les arguments de performance sont donc repoussés à l'extrême du spectre. De nombreux autres problèmes empêchent le paradigme agent de

se populariser, notamment les problèmes de communication entre agents et la maintenance des agents.

Les difficultés spécifiques des agents intelligents sont similaires à celles de l'Intelligence Artificielle : partage des connaissances, ontologies et communications, entre autres. Plusieurs applications sont développées et qualifiées applications multi-agents. Cependant, après un examen approfondi, on s'aperçoit que ces systèmes utilisent plusieurs agents uniquement pour augmenter la rapidité, la fiabilité (redondance), la vitesse et l'efficacité de l'application. Un système est accepté comme un système multi-agent si l'utilisation séquentielle d'un seul agent ne permet pas d'obtenir les mêmes résultats. La raison d'être d'un système multi-agent est que l'interaction des agents augmente globalement l'efficacité du système et lui permet d'accomplir des tâches non réalisables autrement. Autrement dit, l'ensemble doit être plus grand que la somme de ses parties. Objectivement, peu d'applications multi-agents ont déjà été développées et à notre connaissance aucune application multi-agent ouverte n'a encore vu le jour.

Les difficultés spécifiques des agents mobiles sont liées en général au manque de normalisation. Le déploiement des applications mobiles a souffert du manque d'environnement d'exécution et d'hébergement pour agents. Heureusement, la présence de ORBs et d'une machine virtuelle Java dans chaque fureteur (Mosaic, Netscape, I.E.), pourrait combler ce vide pourvu que des interfaces ou plugiciels pour code mobile soient développés.

Malgré ces difficultés, de plus en plus d'applications mobiles sont développées. Ceci mène à un développement en spirale où les standards, les systèmes et les applications sont tour à tour développés et améliorés. Dans la mesure qu'ils permettent de résoudre plusieurs problèmes différents de façon uniforme, les agents devraient s'imposer comme un modèle incontournable.

Des applications ambitieuses comme "Seamless Messaging" et Téléphone Intelligent (qui sera constitué d'un ensemble d'agents préprogrammés) sont conçus pour fonctionner avec des agents. Il est certain que ces applications seront en fonctionnement dans un futur proche. La grande erreur de la communauté des agents mobiles a été de

trop se concentrer sur des questions techniques comme la migration partielle ou le transfert des processus. Il aurait été bénéfique de consacrer plus de temps à bâtir des applications utiles qui auraient servi à populariser le paradigme.

Les agents mobiles sont actuellement utilisés pour accomplir des tâches répétitives et limitées. Il est quelques fois nécessaire d'augmenter les capacités de l'agent pour qu'il puisse réaliser plus de tâches. Les techniques de programmation orienté-objet imposent que l'agent implante ou modifie une interface particulière. Les agents intelligents peuvent apprendre de nouvelles situations ou leur base de connaissances peut être enrichie. En effet, les agents mobiles peuvent obtenir plus d'autonomie en s'inspirant des systèmes à base de connaissances. Cette synergie potentielle justifie une nouvelle méthodologie de développement des agents mobiles : commencer avec un prototype qui démontre les avantages de la mobilité et progressivement ajouter de l'intelligence dans ce prototype.

L'avenir est donc prometteur. Des technologies comme CORBA, XML et Java servlets s'intègrent bien avec les agents. CORBA mobile est déjà une réalité (www.jumpingbeans.com). Les agents mobiles gagneraient à intégrer les méthodes utilisées pour développer les agents intelligents. Les capacités d'un agent mobile peuvent notamment être améliorées s'il est capable d'apprentissage. Les agents mobiles peuvent ainsi être conçus pour avoir beaucoup plus d'autonomie.

Les techniques de mise à jour dynamique présentes dans la littérature ne sont pas satisfaisantes dans la perpective de la mise à jour dynamique d'un agent. Les problèmes proviennent notamment des dépendances avec une plate-forme particulière et du manque de flexibilité et de contrôle des systèmes repertoriés. Il sera donc nécessaire de trouver une solution aussi générique que possible (on devra servir plusieurs types différents de services), qui ne nécessite aucun support du système (en clair, une librairie), et qui occupe aussi peu d'espace mémoire que possible, ceci dans la perspective de la mise à jour sur les moniteurs PDA. Puisqu'elle sera conçue indépendamment des services, cette solution sera alors applicable à n'importe quel logiciel informatique, peu importe son modèle (distribué, statique, agent mobiles, etc.).

CHAPITRE III

MODÉLISATION D'UNE ARCHITECTURE D'AGENT MOBILE POUR SERVICES

Un agent mobile pour services (AMS) est un médiateur qui coordonne et transporte des services auxquels l'usager s'est abonné. L'agent peut transporter les codes exécutables des services ou seulement des pointeurs aux codes. Une architecture générique et extensible est essentielle pour un tel agent puisqu'il peut contenir un grand nombre de services divers. De plus, cet agent doit être conçu pour supporter un ensemble augmentable d'opérations. Ce chapitre traite de la modélisation d'une architecture d'agent mobile pour service. Pour commencer, nous examinons les propriétés que doit avoir notre architecture et énumérons les caractéristiques des AMS que cette architecture devrait permettre de créer. Nous présentons ensuite notre architecture, ses composantes et leurs interfaces. Par la suite, nous détaillons un AMS générique en précisant ses opérations de base, ses patrons de design, son modèle architectural, son encapsulation des connaissances qui lui permet de coordonner ses services. Pour finir, nous présentons une implantation de notre architecture et indiquons des directions à explorer pour améliorer notre implantation.

3.1 Caractéristiques de l'architecture de service et de l'AMS

L'architecture de service que nous proposons est une version modifiée de celles proposées par Glitho *et al.* (2000), spécifiant un AMS par service ou un AMS unique pour tous les services. Notre architecture propose un AMS par classe de services. L'AMS peut alors être conçu spécialement pour des services de téléphonie ou pour des services non reliés à la téléphonie. Dans ces groupes, il peut même cibler des classes de services particulières (outgoing call screening, incoming call screening). De plus, nous introduisons ici une analyse pour la communication inter-agent dans un tel contexte. Pour

finir, l'effort décrit ici est la première implémentation connue d'un AMS. Glitho *et al.* (Infocom 2000) détaillent le concept et ses possibilités. La pratique nous a permis de compléter la description du concept, d'y apporter certaines corrections et de le valider.

Une architecture adéquate pour la mobilité des services doit prendre en compte l'ensemble des contraintes inhérentes à la provision de services dans un environnement distribué. En plus de la mobilité des services, le système doit notamment permettre :

- à l'usager de souscrire ou d'annuler des abonnements à un nombre extensible de services ;
- de configurer des AMS dédiés pour des classes de services précises ;
- d'assurer un délai minimal entre l'abonnement et la provision du service sur le terminal de l'usager ou au nœud du réseau ;
- de gérer le profil de chaque usager (abonnements, AMSs associés à cet usager) ;
- la création et l'ajout dynamique de services dans le système ;
- d'assurer la maintenance à distance des services coordonnés par un AMS donné.

Généralement, les usagers souscrivent à des services ou les annulent de façon très dynamique. Il est donc nécessaire de répartir les fonctionnalités du système entre divers blocs qui accomplissent leur tâche particulière de façon optimale. Ces blocs sont intégrés lors de la réalisation de chaque transaction. Cette approche augmente la performance du système et sa fiabilité. En effet, puisque les blocs ne forment des liens que lors du traitement d'une requête, chaque bloc continuera à opérer indépendamment des autres si un ou plusieurs blocs ne sont pas disponibles (panne, faute matérielle, etc.).

Les principales fonctionnalités du système sont : l'ajout et la création de services, la publication de ces services, la gestion des abonnements (profil usager, etc.), la création et la maintenance des AMS. Ces fonctionnalités peuvent donc être reparties comme suit :

- une unité de création de services (UCS) ;
- une unité de gestion du système (UGS) qui se chargera de la gestion des usagers, de la création des AMS et de leur maintenance ;
- une unité de publication des services (UPS).

L'unité de création des services intègre un environnement pour la création des services, un serveur de fichiers et une base de données de services. L'unité de gestion des usagers se charge de la création des AMS car ceux-ci sont assemblés différemment selon les préférences de l'usager. Théoriquement, la création des AMS aurait pu être assurée dans une autre unité. Toutefois, pour tirer profit de la localité des données sur l'usager qui est indispensable pour modeliser l'intelligence qui sera fournie au AMS, nous avons préféré regrouper ces fonctionnalités dans le même bloc. L'unité de publication des services utilise une interface usager par laquelle les abonnés souscrivent aux services. Elle transmet les requêtes à l'unité de gestion des usagers.

3.1.1 Architecture de l'AMS

L'AMS doit permettre à l'usager de démarrer localement tout service auquel il a souscrit. Il doit aussi lui permettre d'arrêter et de déplacer ces services. Pour accomplir ces tâches, il doit offrir une interface usager par laquelle l'usager interagira avec les services. En fait, l'AMS doit être conçu de sorte que l'usager ne perçoive aucune différence entre ses services et des programmes locaux. L'usager n'est donc pas supposé savoir que les services sont coordonnés par un AMS. De plus, l'AMS doit prévoir des mécanismes pour la mise à jour dynamique du code. Il doit donc implanter des interfaces ou supporter un langage et une ontologie définis pour le transfert et la sauvegarde de données personnalisées (éventuellement modifiées par l'usager) relatives à chaque service. Il doit être conçu de manière à pouvoir transférer et exécuter du code distant.

La mise à jour dynamique du code est basée sur l'héritage de classes abstraites. L'AMS se constituera en libraire pour cette opération. Les primitives du système d'exploitation ne pourront pas être utilisées.

L'architecture de l'AMS doit aussi prévoir la seconde stratégie de maintenance des services, celles de l'échange d'agent (un agent contenant les anciens et nouveaux services vient remplacer l'ancien agent qui contenait seulement les anciens services). Il doit opérer indépendamment de la plate-forme pour agents mobiles qu'il utilise. Ceci assure sa portabilité.

L'AMS utilisera un langage de communication inter-agent (KQML) et implante l'ontologie (Ontologie-AMS) définie pour cette application. L'AMS définira une politique d'accès aux ressources pour chaque service afin d'offrir une sécurité minimale durant l'exécution desdits services.

De façon générale, l'AMS sera composé des codes des services (ou pointeurs au code) et de l'intelligence (logique d'invocation) pour démarrer et coordonner ses services (données personnalisées). Les problèmes sur lesquels cette maîtrise s'attarde sont : la définition de l'architecture et la maintenance des services. Des exigences pour des stratégies élaborées sur l'authentification de l'AMS et sa sécurité seront formulées dans le cadre de recherches futures. Comme pour toute application distribuée, la sécurité est une question cruciale qui doit être aussi prise en compte.

3.2 Architecture des unités du système

Les unités du système que nous examinons ici sont : l'UGS, l'UPS (dans notre implémentation, un serveur Web) et l'UCS. Ces serveurs interagissent pour la provision de services à l'usager. Une implémentation typique du système est présentée à la Figure 3.1.

Chaque usager est attaché à une seule et unique UGS. Les UPS savent quel UGS correspond à chaque usager et lui transmettent l'information relative à cet usager (abonnements, etc.). À la Figure 3.1, l'usager se connecte à une UPS. Celle-ci propage l'information relative à l'usager à son UGS (abonnements, identification, adresse de la machine). L'UGS envoie un AMS à la machine cible où à un noeud du réseau dans le cas de service de téléphonie. Chaque UCS informe un UGS quand un nouveau service est ajouté dans sa base. Les unités UGS, UPS et UCS contiennent des serveurs qui sont répliqués. Cependant, aucune paire de ces unités (deux UPS, UGS, UCS) n'est identique.

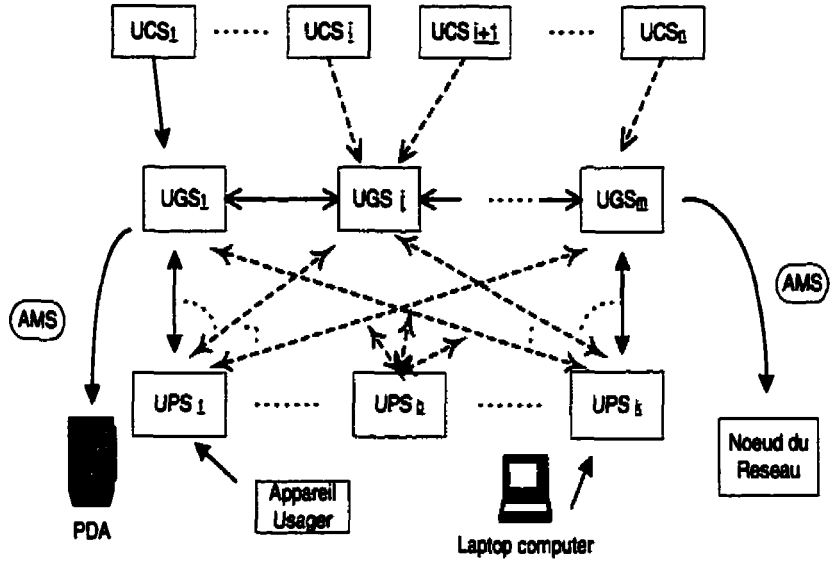


Figure 3.1 Architecture de service par agent

3.2.1 Interface UCS-UGS

Quand un nouveau service est ajouté au système, l'UCS concernée ouvre une connexion TCP/IP (protocole HTTP) avec une UGS et l'informe de l'ajout du service. Durant la connexion, l'UCS transmet son identification (localisation), le nom du service qui a été ajouté, la liste des fichiers du service, le point d'entrée du service (parmi tous les fichiers) et le nombre de paramètres obligatoires à l'invocation. Les paramètres de départ sont des chaînes de caractères. Cette information est alors répliquée atomiquement dans les autres UGS.

Le protocole HTTP version 1.1 est utilisé pour démarrer, arrêter la connexion et spécifier la taille de l'information transmise. L'entête d'une connexion ressemble typiquement à l'entête présentée à la Figure 3.2.

```
HTTP/1.1 200 OK
Date: Fri, 30 Oct 1998 13:19:41 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Mon, 29 Jun 1998 02:28:12 GMT
Content-Length: 1040
Content-Type: text/AMS
```

Figure 3.2 Entête HTTP

Le champ "content-length" donne la longueur de la chaîne de caractère qui décrit le service. Cette chaîne est formatée comme suit : la première ligne contient l'adresse de l'UCS; la seconde ligne conserve le nom du service; la troisième ligne donne la liste des fichiers du service; la quatrième ligne indique le point d'entrée du service; la cinquième ligne donne le nombre de paramètres de démarrage. La figure 3.3 en est une illustration.

```
Location: 142.133.XX.XXX \n
Service: Meeting Planner \n
Files: MobileAgent.Secretary.Assistant.class File1.txt etc. \n
Entry_Point: MobileAgent.Secretary.Assistant.class \n
Starting_Parameters: 2 \n \n
```

Figure 3.3 Format de description de l'ajout d'un nouveau service

L'interaction entre l'UCS et l'UGS pour l'ajout d'un service est décrite à la Figure 3.4.

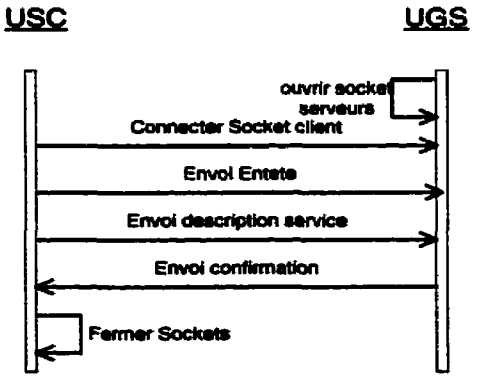


Figure 3.4 Interaction UCS-UGS pour l'ajout d'un nouveau service

L'UGS accepte des connexions avec l'UCS au port 5000. À l'ajout d'un nouveau service, l'UCS ouvre une connexion HTTP avec le premier UGS disponible. Cette UGS propage l'information aux UGS miroirs comme l'illustre la Figure 3.5.

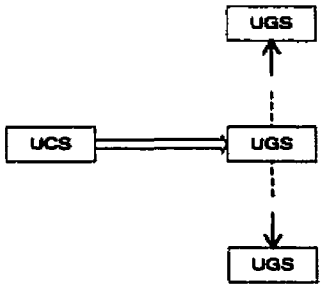


Figure 3.5 Ajout d'un nouveau service

3.2.2 Interface UGS-UPS

Chaque UPS possède l'adresse de toutes les UGS et chaque UGS a aussi l'adresse de toutes les UPS. Quand une UPS veut informer une UGS d'une nouvelle souscription, il contacte l'UGS à laquelle l'usager est attaché. Des schémas alternatifs seraient que les usagers soient assignés aux UGS en ordre alphabétique ou par localisation géographique. Dans le premier cas, les abonnés dont le nom commence par une lettre comprise entre A et D sont assignés à l'UGS_1 ; ceux entre E et H sont confiés à l'UGS_2, et ainsi de suite. La Figure 3.6 illustre l'interface UPS-UGS.

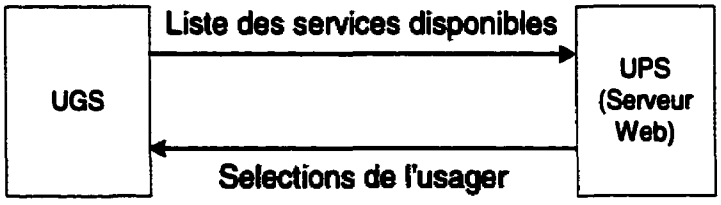


Figure 3.6 Interface UPS-UGS

L'UPS se sert de l'identification de l'abonné pour déterminer son UGS. Les deux unités communiquent à l'aide du protocole HTTP. Pour l'instant, ils utilisent les mêmes champs d'en-tête que l'interface UGS-UCS. L'information échangée est aussi formatée de la même façon. Le contenu est toutefois différent. Il suit le format présenté à la Figure 3.7.

```
Location: 142.133.XX.XXX \n
User_ID: Identification_de_l'usager \n
Requested_Services: Service1 Service2 \n
Target_Location: Ou_Envoyer_AMS \n\n
```

Figure 3.7 Information transmise lors d'un abonnement

Le champ Target_Location est optionnel. Lorsqu'il est omis, le défaut est la machine actuelle de l'usager ou, pour certains AMS, le nœud du réseau où les services de téléphonie de cet usager doivent résider.

3.2.3 Architecture de l'UGS

L'UGS est un serveur "multithread" qui supporte plusieurs connexions HTTP concurrentes. C'est un serveur orienté objet qui implémente le modèle de la composition des classes. L'UGS

- maintient des bases de données d'abonnés et leurs services, gère un sous-ensemble des abonnés ;
- propage l'information sur l'ajout d'un service aux autres UGS ;
- crée des AMS et construit l'intelligence qui leur est nécessaire pour transférer et exécuter les services ;
- informe les UPS de l'ajout de nouveaux services.

Nous présentons d'abord le protocole de communication que le SMU utilise. Nous regardons ensuite la création d'un AMS. Finalement, nous présentons les mécanismes en place pour assurer la sécurité et la fiabilité de l'UGS.

Protocole de communication

L'UGS communique par connexions sockets et implante le protocole HTTP. Les ports 5000 et 7000 sont respectivement réservés pour les communications avec l'UCS et l'UPS. Dans le cas où ces ports seraient utilisés, on peut communiquer avec l'UGS par le port 5000. Il ouvrira alors une connexion avec le client au port convenu.

Création de l'AMS

Dans le cas le plus simple, le système est composé d'une UGS, d'une UPS, d'une UCS, d'une machine cible et d'une machine client. L'AMS est envoyé à la machine cible ou à un nœud du réseau selon les paramètres de configuration de l'UGS. La machine cible ou le nœud du réseau doit avoir un environnement d'exécution pour agents mobiles. L'interaction la plus simple pour la création d'un AMS est montrée à la Figure 3.8.

1. L'usager se connecte à l'UPS et s'abonne à des services;
2. L'UPS contacte l'UGS appropriée et lui passe la liste des services et l'adresse de destination de l'AMS;
3. L'AMS est envoyé à la machine cible qui peut être la même que la machine client.

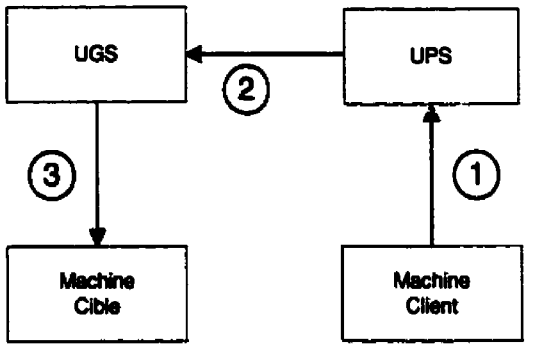


Figure 3.8 Interaction pour la création de l'AMS

La classe principale de l'AMS est une des composantes de l'UGS. Cette classe est un réservoir de connaissances et contient l'intelligence de l'AMS. L'AMS est créé en assemblant son intelligence et la liste des pointeurs aux services. L'UGS crée une instance de l'AMS et lui passe l'intelligence pour la gestion des services. Cette intelligence permet à l'AMS de charger, démarrer et arrêter chacun de ses services.

À ce stade de l'implantation, les actions intelligentes de l'AMS sont :

- mettre à jour les services lorsque l'abonné change ses souscriptions ;
- offrir un menu personnalisé pour le démarrage de chaque service ;
- sauvegarder les données personnalisées de l'usager, relatives à chaque service et les maintenir à travers les versions.

L'AMS peut prendre le code des services avant de se rendre sur la machine cible, ou il peut les charger sélectivement à son arrivée. La stratégie adoptée dépendra de l'espace disponible sur cette machine cible. Ceci est critique pour les moniteurs tels les "palmtop" et autres appareils dont les processeurs et l'espace de stockage sont limités.

Fiabilité et sécurité

La liste des services disponibles est la même dans toutes les UGS. Les UGS sont toutefois différentes car l'information relative à un abonné est conservée à une seule et unique UGS. Une UGS qui gère un sous-ensemble d'usagers peut être répliquée. Cette redondance augmente la fiabilité du système. Ces unités répliquées peuvent être dispersées géographiquement pour réaliser une couverture efficace. Les autres données disponibles à une UGS sont répliquées à tous les autres UGS. Les mises à jour entre USGs doivent être atomiques. Comme l'illustre la Figure 3.9, la liste des services disponibles est échangée entre groupes repliqués d'UGS.

Les communications avec l'UGS sont encodées. Une voie intéressante serait d'utiliser SHTTP (Secure-HTPP). Malheureusement, les développements du protocole et son adoption comme standard sont actuellement arrêtés. Toutefois, l'objectif demeure d'implanter une solution semblable à SSL (Secure Socket Layer).

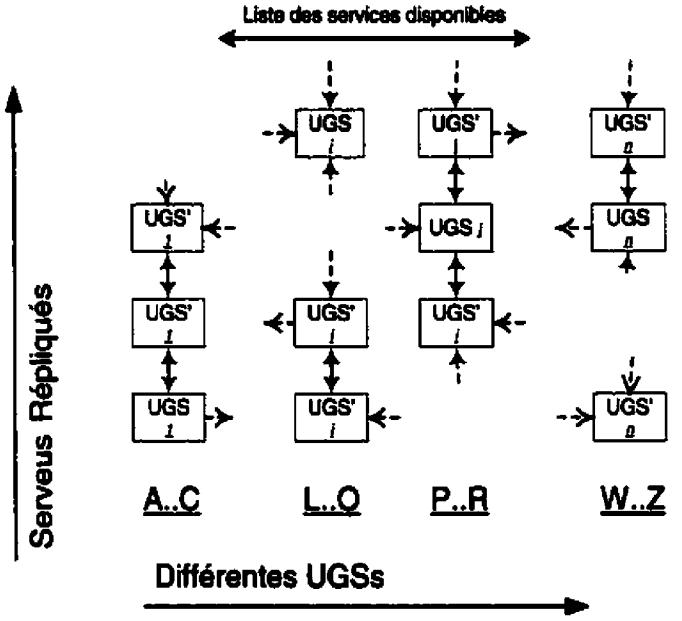


Figure 3.9 Réplication d'UGS pour un sous-groupe d'usagers

3.2.4 Architecture de l'UCS

L'UCS offre un environnement de développement des services (EDS). Cet environnement doit être utilisé pour simuler l'exécution des services avant qu'ils ne soient transportés par un AMS. L'UCS est aussi constitué d'une base de services qui conserve le code exécutable des services développés dans l'EDS. Le dernier composant de l'UCS est un serveur de fichiers basé sur le protocole HTTP, qui transmet les fichiers disponibles dans la base de service, à travers le réseau aux AMSs. Les fichiers dans la base de services sont divisés selon les services auxquels ils appartiennent.

L'EDS est essentiellement constitué d'une collection de librairies Java (pour tester les services) et d'un guide de programmation pour la construction des services. En effet, les services qui seront transportés par un AMS doivent être autosuffisants, c'est à dire un service ne peut en invoquer un autre et les interactions inter-services sont interdites. L'AMS constitue le lien qui permet d'assembler les services en un ensemble

cohérent. Cette exigence permet d'éliminer les dépendances inter-services et inter-fonctionnalités. Les usagers sont donc libres de souscrire à un ensemble arbitraire de services, sans que la présence de l'un n'impose celle de l'autre.

Ces restrictions favorisent un meilleur modèle de programmation, avec une séparation effective des tâches entre services. Elles contraignent aussi une certaine spécialisation : à chaque fonctionnalité précise doit correspondre un service indépendant. Ceci est particulièrement vrai dans le contexte des réseaux intelligents, où les faiblesses du système proviennent des dépendances qui se créent entre services (Yin *et al.*, 1993). Qui plus est, le nombre de services qu'on ajoute au réseau (transfert d'appel, etc.) va grandissant complexifiant toujours plus la gestion des services et de leurs interactions (Jackson *et al.*, 1998). Le remodelage des anciennes fonctionnalités pour accommoder les nouvelles n'est pas une bonne option en pratique.

Les restrictions imposées sur les services sont destinées à faciliter leur maintenance et à augmenter la productivité de leur interaction. La complexité des décisions relatives à leur fonctionnement est localisée dans l'AMS.

3.3 Architecture de l'Agent Mobile pour Services

L'AMS est caractérisé par l'ensemble des opérations qu'il supporte. Il doit notamment :

- charger le code des services ;
- démarrer ses services à la demande ;
- se déplacer sur demande de l'usager ;
- permettre la mise à jour dynamique (sans interruption) de ses services ;
- être remplaçable ;
- prévoir des mécanismes de recouvrement de fautes et de sécurité pour ses services.

En plus des opérations spécifiques listées plus haut, l'AMS doit disposer de fonctions de base nécessaires pour évoluer dans une société d'agents, à savoir langage et ontologie, sécurité et tolérance aux fautes. Nous traiterons donc des fonctions de base de

l'AMS dans la section 3.3.2. Dans la section 3.3.3, nous donnons les directives pour le chargement et le démarrage des services. Finalement, dans la section 3.3.4, nous définissons le cadre logiciel nécessaire pour la gestion des services. Nous n'aborderons pas ici les questions de remplacement de l'AMS et de sa mise à jour dynamique. Ces sujets sont putôt traités aux chapitre quatre et cinq. Pour commencer, examinons l'évolution de l'AMS dans le temps.

3.3.1 Évolution de l'AMS

Le diagramme d'état permet de retracer l'évolution temporelle de l'AMS ainsi que ses réactions aux influx externes (choix de l'usager, appel de procédure à distance, dialogue inter-agent). La Figure 3.10 présente le diagramme d'état de l'AMS. On y retrace ses possibles interactions et son évolution depuis sa création à l'UGS, la provision de services au nœud de destination, ses futurs déplacements, son remplacement et sa mise à jour éventuelle.

L'analyse du diagramme d'état permet de remarquer les interactions de l'AMS avec l'extérieur. Il enregistre les données personnalisées de l'usager sur disque. Il converse avec le nouvel AMS pour s'entendre sur les stratégies de remplacement dans le cas d'une mise à jour. Il converse avec un agent de mise à jour dans le cas où le renouvellement doit être dynamique. Celui-ci lui passe alors l'information pour charger les nouvelles versions ou classes des services ciblés. Alternativement, il peut aussi être invoqué par appel de méthode à distance.

L'AMS offre un menu qui permet à l'usager de sélectionner les services à démarrer. L'AMS n'exerce aucun contrôle subséquent sur ces services. Sa seule action possible est d'arrêter leur exécution. Le même menu permet à l'utilisateur de décider s'il veut relocaliser ses services. Il entre alors la machine de destination et l'AMS déplace son clone.

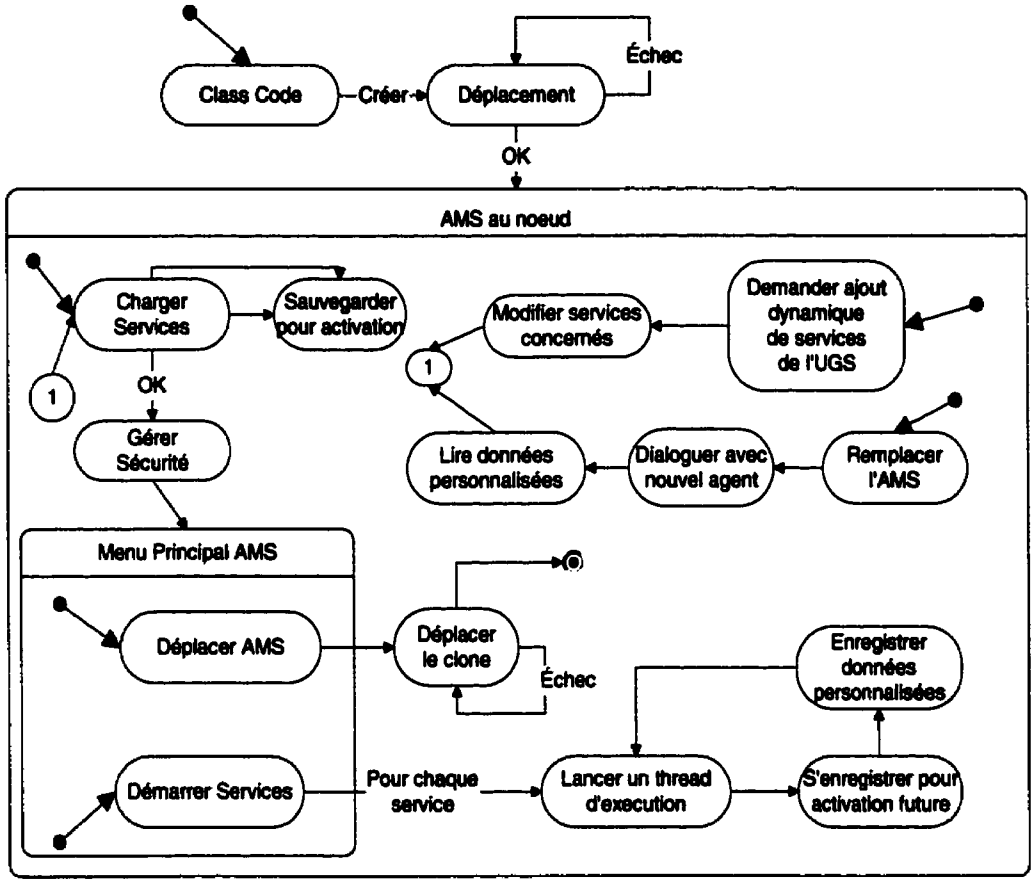


Figure 3.10 Diagramme d'état de l'AMS

3.3.2 Langage, ontologie, mobilité, sécurité et fiabilité

L'implantation des cinq fonctionnalités subira des petites variations selon la plate-forme pour agents mobiles utilisée. Leurs spécifications ici prennent donc soin de limiter les dépendances à la plate-forme de manière à augmenter la portabilité et la réutilisabilité de toute implémentation. Les spécifications indiquent aussi qu'il ne devrait pas exister d'interdépendance entre ces fonctionnalités. Elles sont donc réparties sur cinq couches (une couche par propriété) indépendantes les unes des autres.

Langage

L'AMS communique en KQML de UMBC (University of Maryland at Baltimore County) et utilise la librairie Java KQML (*JKQML*) développée par IBM AlphaWorks (voir site *JKQML*). *JKQML* a été choisie parce que l'implantation FIPA-ACL de *Nortel Networks* demande plus de support machine. De plus, nous ne disposions pas de référence d'implantation réussie en FIPA-ACL qui n'est disponible que pour les machines Unix. Son installation nécessite des privilèges d'administrateur.

Les primitives du langage (performatives) supportées par *JKQML* sont facilement compréhensibles (stop, tell, move, etc.). De plus le format des messages est basé sur la logique des prédicats. La proposition de Labrou *et al.* (1996) pour le nouveau format des messages permet d'étendre le format KQML pour les besoins d'une application précise. *JKQML* permet d'adresser des messages synchrones et asynchrones. Une requête pour le transfert des données personnalisées par l'usager s'écrit par exemple comme indiqué à la Figure 3.11.

```
try{
        msg.setPerformative(KQML.TELL) ;
        msg.setSender(this.getAddress()) ;
        msg.setReceiver(adresse_de_agent_a_contacter) ;
        msg.setOntology(MSA_Service_Ontology) ;
        msg.setLanguage(« KQML ») ;
        msg.setContent(«Passer donnees personnelles sous format
                        Service : nom_service
                        Donnees : options modifiees
                        dans la reponse ») ;
        msg.setIRT(none)  // IRT = In Reply To
}catch (Exception e) {
...
}
```

Figure 3.11 Envoi de messages KQML

Il suffit d'installer un gestionnaire KQML à chacun des hôtes pour qu'il transmette les messages à l'agent. Concrètement, un message a le format présenté à la Figure 3.12.

```
( achieve // pourrait être tell, send, find, etc.
        :sender demandeur
        :receiver destination
        :reply –with id_personnelle
        :language LISP  // pourrait être Java ou Scheme
        :ontology MSA_Service
        :content chaine_de_caractère_décrivant_le_contenu_spécifique_du_message )
```

Figure 3.12 Format du message envoyé selon le protocole KQML

Ontologie

Modéliser une ontologie, c'est essayer d'établir une base formelle et concise pour la communication, le partage de connaissances et l'interprétation de messages dans un domaine précis. La première exigence est de s'assurer de la cohérence (non-ambiguïté) du formalisme. On doit donc prouver cette cohérence en utilisant une logique précise (prédicats du premier ordre le plus souvent) et en se basant sur des axiomes connus (axiomes de la logique cartésienne généralement). Il faut aussi définir le format du langage et fournir un premier dictionnaire de termes pour le domaine d'application ciblé.

Nous avons survolé les considérations théoriques pour définir une ontologie restreinte, limitée à la communication inter-AMSs et AMS-agent de mise à jour. Cette ontologie est un sous-ensemble de KIF (voir site KIF Specification) qui est une ontologie générale jouant le rôle de médium entre deux ontologies traitant de sujets différents. KIF a été choisi à cause de l'existence d'un analyseur de texte KIF à Java : Java KIF Parser (*JKP*), disponible gratuitement (voir site *JKP*). Une fois que la chaîne KIF a été transformée en objet Java, on utilise *Xerces* (anciennement XML To Java de IBM AlphaWorks) de Apache.org pour obtenir une représentation XML du contenu du

message. L'AMS peut donc communiquer avec des agents provenant de vendeurs quelconques. Ceci est possible parce que KIF est indépendant du langage naturel utilisé (français, anglais, bamiléké, etc.). Une bonne référence sur la modélisation d'ontologie est le document de Gruber (1993).

Un dialogue KIF référant à une quantité physique quelconque est encodé comme illustré à la Figure 3.13. Il y manque notamment l'unité de mesure de la quantité physique ainsi que d'autres paramètres.

```
(defrelation QUANTITÉ-PHYSIQUE
  ( ⇔ (QUANTITÉ-PHYSIQUE ?q)
        ( et (définie (magnitude de l'unité ?q) )
             (magnitude(magnitude de l'unité ?q) )
.... // plusieurs autres lignes pour s'assurer de la cohérence de l'énoncé
```

Figure 3.13 Exemple d'une formulation simple en KIF

Dans le contexte de l'AMS, les agents communiquent par exemple pour s'échanger des données modifiées par l'usager comme illustré à la Figure 3.14.

1er agent	(interested service '(service , ?service1, ?service2, service3))
2e agent	(service1 modif1 paramaters)
	>(service2 ?x) > 8
	etc.

Figure 3.14 Échange de renseignements sur un service

Le nouvel AMS demande de l'information sur les services 1, 2 et 3. L'ancien lui dit que le service1 doit être invoqué avec les paramètres modifiés (parameters). Le service 2 doit être invoqué avec des valeurs supérieures à huit. Ce sont ici des exemples simples. Nous nous sommes limités à ces messages simples pour notre première implémentation. Bien que l'apprentissage soit pénible, il est préférable d'implanter des agents qui peuvent converser dans une ontologie précise. Il n'est pas toujours facile

d'exprimer toutes les relations (combien de fonctions faudrait-il prévoir pour les cas où les valeurs sont supérieures, inférieures et modifiées). De plus, des logiciels gratuits pour l'analyse de texte sont disponibles (voir site *XML-JAVA* et *JKP*).

Mobilité, sécurité et fiabilité

La mobilité est pourvue par la plate-forme pour agents mobiles. Toutefois, dans le cadre de notre application, nous avons modifié la couche Transport de *Voyager* pour accélérer le transfert de données et du code objet de certains fichiers quand l'AMS se déplaçait. Ceci n'est théoriquement pas nécessaire et n'a été fait que parce que nous avons remarqué des fautes dont nous ne pouvions retracer l'origine.

L'AMS implante les mécanismes d'authentification disponibles par l'interface Java.Security.* de JDK1.2.2. Malheureusement, nous n'avons pas pu les tester extensivement, manquant de temps et étant limité à notre laboratoire pour l'implantation. Des outils existent aussi en Java pour le codage et le décodage de données (MD5, SHA). Le design de l'AMS a été fait pour utiliser ces classes suivant l'usage qui en a été fait pour les communications UCS-UGS. Toutefois, ces classes n'ont pas été utilisées; l'UCS et l'UGS les utilisent de façon simpliste sur de faibles quantités de données. Des erreurs surprenantes ont émergé lors de l'encodage et du décodage des fichiers objets.

En effet, le premier design prévoyait qu'on chargeait les fichiers objets des services, les encryptait, puis on les enregistrait sur le disque. Ceci avait trois bénéfices : robustesse, rapidité et sécurité. L'usager ne pouvait utiliser directement les services puisqu'ils étaient codés et seul l'AMS avait la clé (sécurité). En cas de panne de la machine, l'AMS n'avait pas à recharger les fichiers des services (robustesse). Quand un nouvel agent venait remplacer l'ancien, il n'avait pas à recharger les services à son tour. Il suffisait que l'ancien lui passe la clé de codage (rapidité). Nous avons abandonné cette voie quand nous nous sommes rendus compte qu'après avoir codé et décodé des services, il se produisait des erreurs au moment de l'édition de liens dynamiques par l'interface ClassLoader de Java.

La fiabilité est basée sur l'interface *Activation* de *Voyager*. Cette interface permet de rendre un objet persistant. L'état de l'objet (variables et constantes) est enregistré dans un fichier. Plus tard dans le programme, on invoque cet objet et indique la localisation pour l'activer. L'objet correspondant est reconstitué. On n'a donc pas à renvoyer l'AMS à la machine de l'usager après chaque redémarrage de l'ordinateur. Une fois envoyé, l'objet s'enregistre pour activation. Le problème est qu'il faut écrire une routine pour réactiver l'objet. Nous avons contourné cette difficulté en faisant en sorte que l'AMS enregistre *Voyager* comme un service avec l'AMS comme extension sur chaque machine. On s'est servi de l'utilitaire srvany.exe (disponible sur toute machine Windows NT 4.0) pour y parvenir. Cet utilitaire met à jour les registres de Windows pour opérer. Cette solution n'est donc pas pratique pour un contexte d'utilisation réel. Lors du déplacement de l'agent, un clone est d'abord envoyé. Lorsqu'il arrive à destination, il en informe l'AMS original. Celui-ci s'auto-détruit alors. Il est donc rare qu'on perde un agent au cours d'un déplacement.

3.3.3 Chargement et démarrage des services

L'AMS transfère les services des UCS en ouvrant une connexion socket avec l'UCS. Il utilise le protocole HTTP pour le transfert des fichiers. N'importe quel langage qui permet une édition des liens dynamiques (Oberon, LISP, Smalltalk, etc.) au moment de l'exécution du code peut être utilisé pour alors charger et exécuter les services. Notre implémentation s'est faite en Java. Les exemples sont des extraits simplifiés de notre code. L'interface ClassLoader (Figure 3.15) de Java permet de charger et d'exécuter n'importe lequel des fichiers objets à distance.

```
public abstract class ClassLoader {
    public Class loadClass(String name);
    protected Class findClass(String name);
    ...
}
```

Figure 3.15 ClassLoader JDK 1.2.2

On se sert donc de cette interface pour charger et procéder à la résolution dynamique des liens des services comme illustré à la Figure 3.16.

```
class MSAClassLoader extends ClassLoader{
   String scu_location;
   public MSAClassLoader (String location) {
            scu_location = location;
        }
   protected Class findClass(String name){
            byte[] classbytes = getClassCode(name);
            return(defineClass(name,classbytes,0,classbytes.length));
   }
   public Class loadClass(String name) {...}
   byte[] getClassCode(String name) {...}
   byte[] getClassFromArchive(String name) {...}
   ...
}
```

Figure 3.16 ClassLoader AMS

Une fois l'édition de lien terminé, on se sert de l'interface Reflection de JDK 1.2.2 pour obtenir des instances de la classe et pour démarrer le service. Ceci est illustré à la Figure 3.17.

Java est le langage de choix, car *Voyager* (la plate-forme de développement du prototype) l'utilise. Comme mentionné plus haut, l'édition dynamique de liens permet de résoudre les références aux modules externes au moment de l'exécution. Nous nous étendrons plus longuement sur les éditeurs de liens dynamiques dans le chapitre 5 où nous traitons de la mise à jour dynamique de l'AMS. Le chargement dynamique de code est une partie importante de la solution pour la mise jour dynamique.

```
class MSAServiceProcess extends Thread
{
   String service;
   Object Data; String protocol;
  public MSAServiceProcess (ThreadGroup th, String service, Object
sdata) {
      super(th,service);
      Data = sdata;
        }
        void addServiceClasses(String className){...}
  Object getServiceParam(...){...}
        void run() {
              getServiceRules();
              if(condition 1){
                  launchService(msa, service1)
          }
          else{
          ...
          }
        }
  void launchService(MSA msa, String mainclass){
     MSAClassLoader ms = new MSAClassLoader(loc);
     Class c = ms.loadClass(mainclass);
     Class[] carray = c.getInterfaces();
     If (carray[i].getName != "IMSAService"){...}
     Object serv = c.newInstance();
     Method m = serv.getClass().getMethod("main", new Class[] {...})
     Object o1 = getServiceParam(...);
     m.invoke(null, new Object[] {o1});
     try{
              sleep();
         }catch(InterruptedException e){
             ....
         }
   }
  ...
}
```

Figure 3.17 Lancement d'un service par l'AMS

3.3.4 Encapsulation des connaissances – Gestion des services

L'AMS permet à l'usager de personnaliser les services qu'il transporte. En effet, l'usager peut se servir de l'interface principale de l'AMS pour entrer des paramètres comme les dates d'exécution régulières pour des services. À cette date précise, l'AMS démarrera le service.

Il est aussi possible de changer les paramètres d'invocation d'un service de manière à modifier le comportement du service. L'AMS démarrera alors toujours le service avec ces paramètres-là qui, dépendemment du service, peuvent changer totalement son exécution.

Toutefois, l'AMS ne maintient aucune référence aux objets internes des services. Sa fonction est de les démarrer avec les données personnalisées et les arrêter éventuellement plus tard. Il n'exerce donc aucun contrôle sur l'exécution des services. De la même façon, les services qu'il contient ne font pas référence à l'AMS et n'ont aucune référence entre eux. Une vue de l'AMS est présentée à la Figure 3.18. où AEE (Agent Exécution Environment) désigne la plate-forme pour exécution des agents.

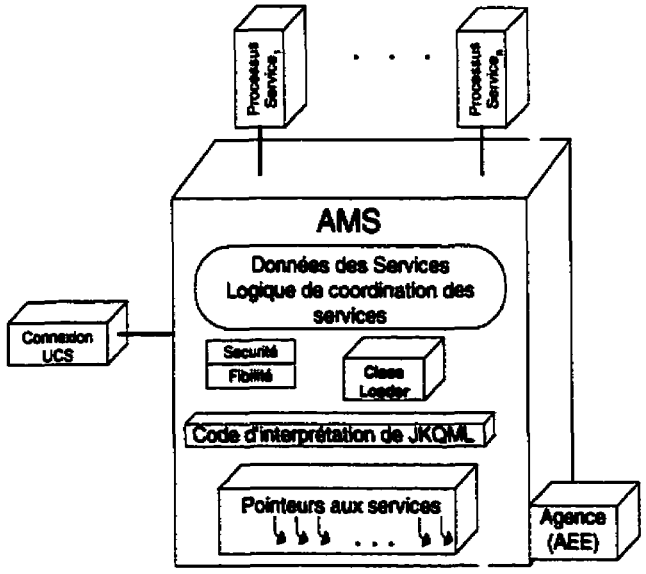


Figure 3.18 Agent Mobile pour Services

3.4 Implémentation préliminaire

Nous présentons ici une analyse structurée basée sur le standard Unified Modeling Language (UML) de l'implémentation. Nous présentons ensuite les applications tests intéressantes qui ont validé notre prototype.

3.4.1 Analyse des interactions avec l'usager

La première interface au système est celle de l'UPS où on peut souscrire à un abonnement. Cette interface est présentée à la Figure 3.19.

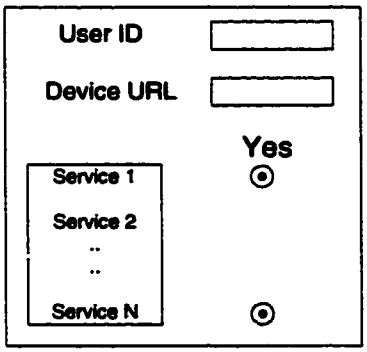


Figure 3.19 Interface d'abonnement

Comme l'illustre la Figure 3.20, le diagramme de contexte du système offre une vue en boite noire du système. Les abonnés ne savent pas que leurs services seront transportés par un AMS. Même lorsque l'AMS est présent sur leur machine, ils ne sont pas tenus de savoir que les services sont offerts et relocalisés par un AMS. Ce qu'ils savent se limite à : je m'abonne à des services. Ces services peuvent être déplacés quand on en fait la demande. On dispose d'une interface pour démarrer les services et changer leurs paramètres d'exécution. En bref, les détails relatifs à l'AMS doivent être transparents à l'usager.

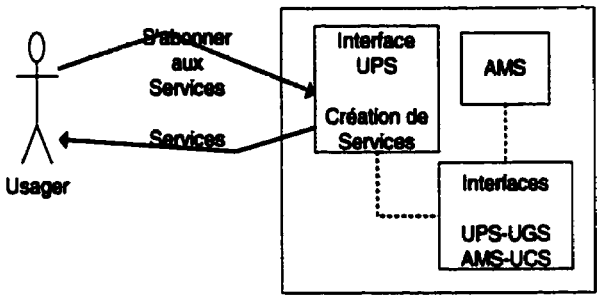


Figure 3.20 Diagramme de contexte

Les cas d'utilisation du système (Use cases) décrivent les possibilités d'utilisations d'un système pour l'usager. Ceux de notre application sont présentés à la Figure 3.21.

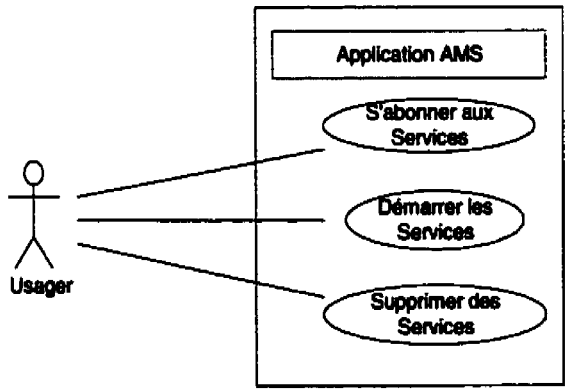


Figure 3.21 Diagramme de cas d'utilisation (Use Cases)

S'abonner aux services

L'utilisateur choisit certains services à partir de l'interface de l'UPS. L'AMS est alors créé et envoyé pour la première fois s'il s'agit d'un premier abonnement. Sinon les

services sont mis à jour selon les stratégies de remplacement de l'agent ou de mise à jour dynamique. Un scénario d'abonnement est illustré à la Figure 3.22.

Démarrer les services

L'AMS sur la machine offre un menu que l'abonné utilise pour lancer un nombre quelconque de services.

Supprimer des services

L'abonné choisit les services à supprimer à partir de l'interface de l'UPS. Un agent portant les messages de suppression de services est envoyé et cet agent fait le tour des AMS pour leur dire lesquels de leurs services ils doivent arrêter.

Les scénarios décrivent les interactions entre les différents acteurs. Même si le rôle de ceux-ci est clair, il est pénible de décrire toutes les interactions possibles. Nous illustrerons ici l'interaction de création de services, car nous pensons qu'elle est essentielle à la compréhension du système.

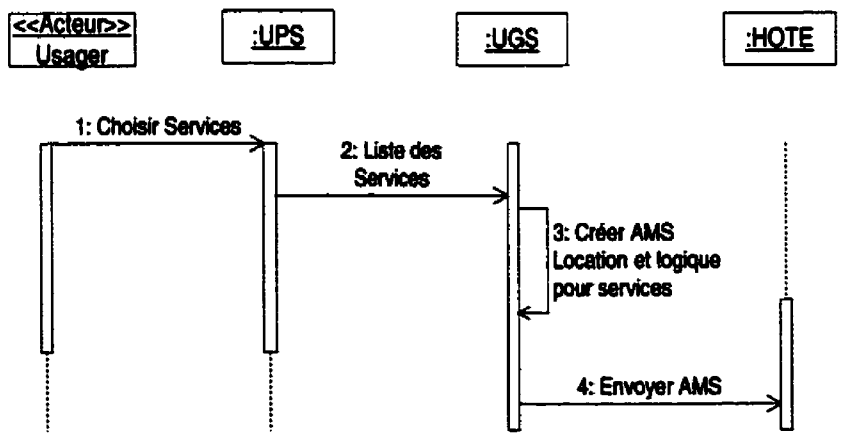


Figure 3.22 Scénario d'abonnement

3.4.2 Analyse de l'UGS

L'AMS est l'unité principale pour la provision de services. L'UGS est l'unité centrale de souscription et de maintenance. Nous présentons ici une partie restreinte mais importante de son diagramme de classe. La Figure 3.23 reflète notre implémentation de l'UGS. De nombreuses classes ont été omises. N'empêche cette vue de haut niveau permet d'entrevoir les interfaces de l'UGS et la création de l'AMS.

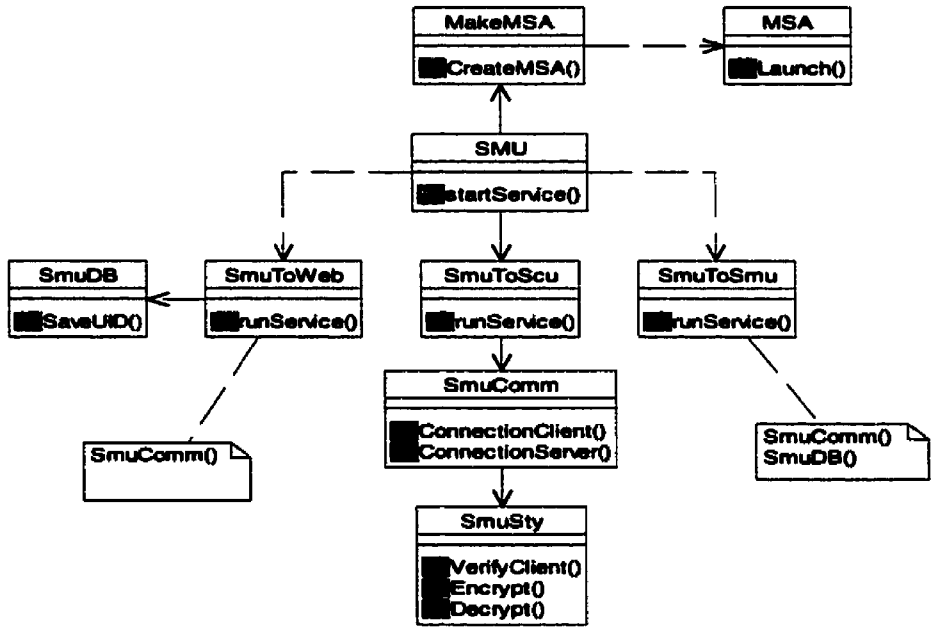


Figure 3.23 Diagramme de classes de l'UGS

Les fonctionnalités de l'UGS sont découplées les unes des autres comme indiqué sur la Figure 3.24. L'UGS est constituée d'une classe principale qui démarre tous ses services : interface avec l'UPS, l'UCS et les autres UGS. Ces services démarrent leur propre Thread de communication. Tous ces Threads de communication utilisent le protocole HTTP et implémente les mêmes mécanismes de sécurité (authentification des partenaires, cryptage des données). Cette sécurité est basée sur l'interface *Security* de JDK 1.2.2.

3.4.3 Analyse de l'UCS

La Figure 3.24 permet d'avoir une représentation visuelle de l'UCS. Elle est composée d'un environnement de création de services et de différents répertoires où ces services sont stockés. L'UCS établit des connexions avec le SMU pour transmettre les nouveaux services et avec l'AMS pour lui servir les fichiers qu'il réclame.

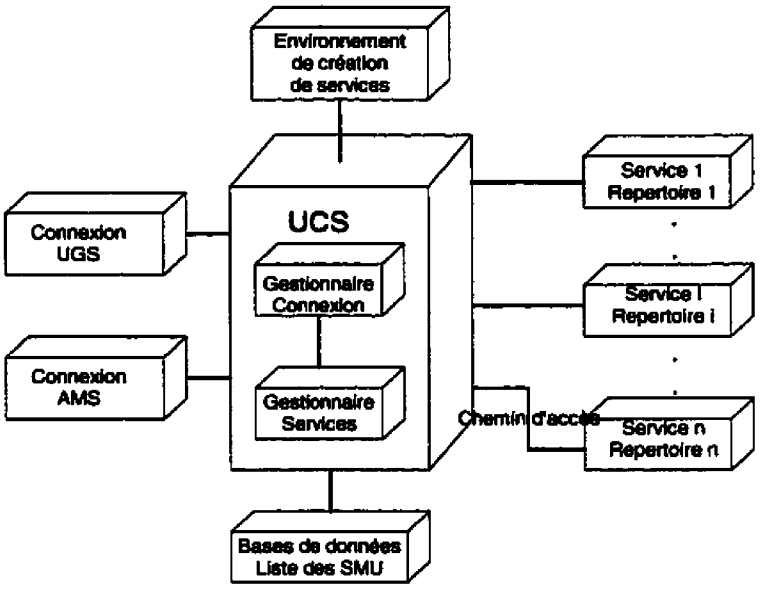


Figure 3.24 Diagramme de l'UCS

3.4.4 Analyse de l'AMS

L'AMS est constitué de :

- une liste extensible de services, et pour chaque service, une liste de ses fichiers ;
- un menu interface graphique qui liste les services et offre l'option de les démarrer arrêter ou déplacer tous ensemble ;
- une classe interne "Thread" (processus poids léger). C'est cette sous-classe qui se charge de lancer les services. L'ensemble des processus poids légers est coordonné par un "ThreadGroup". On peut donc se servir de ce "ThreadGroup" pour avoir des références au contexte d'exécution des services. Chaque Thread (sous-processus du

programme principal qu'est l'AMS) fournit les ressources systèmes nécessaires au service. L'AMS peut donc démarrer un service, l'interrompre temporairement ou définitivement. Un nombre arbitraire de services peut donc être entretenu en même temps.

- une classe interne gestionnaire de sécurité. Cette sous-classe hérite de la classe *SecurityManager* de JDK 1.2. Son modèle de sécurité est donc basé sur celui de Java où on sélectionne les actions possibles du programme selon ses permissions (certificat et authentification). La sous-classe modifie les fonctions du *SecurityManager* et accorde des permissions pour ouvrir et lire des fichiers selon les paramètres qui lui sont passés au moment de la création du service. Une politique de sécurité différente est donc appliquée à chaque service.
- une classe interne qui ouvre une connexion client ou serveur avec l'hôte qui lui est désigné (URL de l'hôte). Cette classe utilise exclusivement le protocole HTTP pour le transfert de fichier. La classe permet de récupérer les "bytecodes" d'une classe d'une UCS distante et tout fichier quelconque qui est nécessaire à l'exécution du service. Toutefois, les fichiers qui ne sont pas du code objet doivent être chargés préalablement à l'exécution et enregistrés sur le disque dur de l'hôte. Les fichiers objets sont transférés au moment de l'exécution, car le ClassLoader de l'AMS permet de les identifier.
- une classe interne (MSAClassLoader) qui, à partir du code objet (bytecodes), opère l'édition de liens et retourne une instance de la classe. La classe interne MSAClassLoader est automatiquement invoquée par la machine virtuelle Java. En effet, quand un programme implémente un ClassLoader, la "Java Virtual Machine" (JVM) invoquera ce ClassLoader quand elle rencontre une référence à un objet ou classe dans le programme. C'est de cette fonctionnalité que nous nous servons d'ailleurs pour tenir compte des versions des services dans notre solution de mise à jour dynamique de l'agent. Vu que c'est notre ClassLoader qui gère le fonctionnement de toutes classes utilisées par l'AMS, il procède en trois étapes. D'abord, elle vérifie que la classe invoquée n'est pas une classe librairie de JDK.

Sinon, elle vérifie si la classe peut être retrouvée en suivant les chemins d'accès indiqués par la variable environnement *CLASSPATH*. En dernier ressort, elle se sert de son intelligence sur la localisation des fichiers de chaque service pour transférer les "bytecodes" de ce fichier d'un hôte distant en utilisant la classe vue plus haut. Il faut noter que JVM ne charge une classe que lors de sa première référence (l'AMS est invoquée une fois pour chaque classe). En effet, dès qu'une classe est chargée, la JVM la garde en cache. Cette version de la classe sera utilisée durant toute l'exécution du programme. JVM assure ainsi l'intégrité du type de la classe. Il faut donc avoir une connaissance profonde de la JVM et des patrons de design pour contourner ces restrictions et installer des classes dynamiques (comme nous l'avons fait pour la mise à jour dynamique).

- une classe interne qui formate, émet et reçoit les messages KQML de l'agent. Cette sous-classe n'est pas difficile à implémenter car elle ne définit pas et n'interprète pas le contenu des messages. Elle spécifie le protocole des messages, varie les paramètres de ces messages selon l'action que l'envoyeur veut que le receveur prenne, et transmet les messages qui sont écrits par la classe MSAOntology. Elle utilise le standard KQML.
- une classe interne MSAOntology qui utilise la logique des prédicats du premier ordre pour spécifier le contenu des messages. Elle utilise le standard KIF. Pour l'instant, elle ne peut coder que trois messages qui indiquent comment les paramètres d'un service ont été modifiés par l'usager, et comment ce service devrait être créé. Nous nous sommes donc servis d'une série de booléens pour indiquer à la classe comment formater le message. La classe ne décode pas les messages (pas de machine d'inférence, il aurait été très pénible de le faire dans le programme). Elle passe ceux-ci à l'utilitaire JKP qui lui retourne un objet (chaîne de caractères) Java plus facilement interprétable. L'utilitaire est installé sur chaque hôte où l'AMS se déplace.

Bien entendu, l'ensemble de ces classes est géré par la logique interne de l'AMS qui associe des fichiers aux services et définit ses actions selon les choix de l'usager.

Plusieurs patrons de design ont été utilisés dans la construction du système. L'UGS implémente notamment les patrons de design Médiateur, singleton et Proxy. Il utilise la composition de classes et programme aux interfaces, et non à l'implémentation de la classe elle-même. Nous reviendrons plus longuement sur ces patrons aux chapitres 4 et 5, car ils sont une composante importante de la solution de mise à jour dynamique.

3.4.5 Services Tests

Deux services d'agents mobiles ont été implémentés pour tester l'AMS. Le premier est un organisateur de réunion. Organiser des réunions entre des dirigeants très occupés signifie souvent que leurs secrétaires doivent passer un temps considérable à essayer de coordonner les calendriers de leurs patrons de manière à aboutir à une date de réunion consensuelle. Même dans le cas simple d'une invitation entre amis, il est souvent difficile de planifier de telles activités.

L'organisateur de réunion envoie un agent avec une sélection des dates disponibles pour la réunion. Cet agent mobile se déplace de participant en participant et recueille chez chacun d'eux sa plage horaire de convenance. Il fait ainsi le tour et revient présenter les dates de rencontre possibles à l'initiateur de la réunion. Si aucune date n'est trouvée, l'agent mobile l'indiquera aussi. En fait, il court-circuitera son trajet dès qu'il se rendra compte qu'il n'y a pas de dates consensuelles après avoir visité quelques participants. Planifier une réunion se fait donc avec un minimum de difficulté.

Le second service cherche le chemin le plus court pour aller d'un point à un autre dans un réseau. Il crée donc plusieurs agents collaboratifs qui disposent tous d'une carte du réseau. Ces agents se répartissent le réseau entre eux et chacun se charge d'en explorer une partie. Le premier qui arrive à destination informe les autres qui rentrent au nœud d'origine. L'agent qui a trouvé la destination revient ensuite. Ce service a une application en téléphonie, car nous avons ainsi simulé le plus court chemin pour traiter un appel. Lors d'un appel, les agents se déplacent sur des nœuds disjoints, invoquent des fonctions de la passerelle pour savoir si elle peut acheminer l'appel. Pour nos simulations, les

agents ouvrent une fenêtre et l'usager répond manuellement s'il permet que l'appel transite par sa machine.

3.5 Synthèse et problèmes ouverts

A ce point-ci, l'implémentation consiste en une UGS, une UPS, une UCS et quelques AMS. Tous les problèmes relatifs à la multiplicité (consistance des données, synchronisation des transactions) des différentes unités n'ont pas encore été traités. Toutefois, les solutions couramment employées pour les systèmes répartis pourraient être appliquées ici : mécanismes de recouvrement de fautes, serveurs redondants pour rediriger les transactions quand le serveur principal n'est pas disponible, etc.

Cette implantation limitée est toutefois suffisante car l'étude concerne la mobilité des services et non les systèmes répartis. Matériellement, l'UGS est située sur une station Solaris 2.5.6 qui a un processeur 400 MHz Pentium 2. L'UCS est située sur une station Windows NT 4.0 avec un processeur Pentium Pro 366 MHz. L'UPS est située sur une machine ayant les mêmes caractéristiques que celle de l'UCS.

Dans l'implémentation, l'UPS est en fait un serveur Web Apache 2.0. Les pages d'abonnement utilisent des Servlets (au lieu de scripts CGI) qui ouvrent des connexions sockets pour transmettre les paramètres de l'abonnement à l'UGS.

L'ontologie implémentée dans le contexte de l'AMS a été définie exclusivement pour son coté pratique : définir la logique et le contenu des messages pouvant être échangés avec l'AMS. Elle n'a pas de fondements théoriques et aucune preuve formelle n'a été effectuée sur sa cohérence. Les primitives du langage utilisables en JKQML sont limitées. L'interprétation des performatives est souvent ambiguë. Toutefois, utiliser un langage présente le grand avantage qu'on n'a pas à changer l'agent si celui-ci doit faire des opérations nouvelles. Si la logique est bonne, un nouveau message devrait susciter les réactions appropriées de la part de l'AMS, ceci même sans machine d'inférence.

La sécurité est faible : de simples algorithmes de codage sont utilisés. Ceci pourrait être rapidement amélioré en utilisant les interfaces que nous avons indiquées plus haut (Voir sites Cryptographie et Sécurité). Les limites de temps ne nous ont pas permis d'installer les mécanismes de sécurité appropriés.

Peu de fonctionnalités dépendent de la plate-forme *Voyager*. Les seules fonctionnalités de *Voyager* que nous utilisons sont le serveur de nom et la possibilité de modifier la couche Transport. Toutefois, le serveur de nom n'est qu'une alternative parmi d'autres pour obtenir une référence à un objet sur une plate-forme Java. Nous l'utilisons au moment du remplacement de l'agent et de sa mise à jour dynamique (chapitre 5). D'autres alternatives sont d'ailleurs présentées au chapitre 5. La modification de la couche est due à des erreurs provoquées par *Voyager*. Nous n'aurions pas besoin de le faire si un autre système était utilisé.

CHAPITRE IV

MISE EN OEUVRE DES SOUSCRIPTIONS

Une architecture novatrice pour la mobilité des services a été présentée et son implémentation spécifiée au chapitre 3. L'agent mobile pour services (AMS) qui transporte les codes exécutables (ou des pointeurs au code) des services est l'entité principale de cette architecture. Transporter des services pose de nouveaux défis relatifs à la gestion de ces services. Ce chapitre examine la mise à jour (ajout, retrait et modification) des services contenus dans un AMS. Intuitivement, il y a deux approches possibles pour la mise à jour d'un AMS : le remplacement de l'agent et sa mise à jour dynamique. Nous analysons d'abord les problèmes posés par la mise à jour d'un AMS, puis nous dérivons un ensemble de requis pour évaluer les solutions possibles. Ensuite, nous présentons l'implémentation de la permutation d'agents. Finalement, nous exposons l'implémentation de la mise à jour dynamique d'un AMS.

4.1 Analyse des problèmes

Tout service pour lequel l'usager a une souscription est transporté par un seul et unique AMS. Pour chaque configuration de l'architecture, le nombre d'AMS par usager est fixé. Nous examinons d'abord le problème de la mise à jour des services et dérivons des requis. Ensuite, nous explicitons les différences entre le remplacement d'agents et la mise à jour dynamique d'un agent.

4.1.1 Problème et requis

Supposons qu'un AMS a été créé pour un usager donné, et que l'AMS contienne les services A, B et C auxquels cet usager s'est abonné. Pour être plus spécifique, l'AMS contient la logique (code exécutable ou pointeurs aux codes), plus les données (ou pointeurs aux données) de chacun de ces trois services. L'AMS gère toujours ses états de façon autonome; il sait donc qu'il contient ces trois services et seulement ces trois

services là. Il est important de noter que les données contenues dans (ou pointées par) l'AMS peuvent être personnalisées par l'utilisateur.

Supposons maintenant que l'usager décide de souscrire à un service additionnel D ou qu'une nouvelle version du service A devienne la norme. La question est de savoir comment l'AMS est mis à jour pour inclure D et la nouvelle version de A en plus de B et C. Deux approches sont possibles : le remplacement d'agent et la mise à jour dynamique de l'agent. Dans la première approche, l'AMS est remplacé par un AMS qui contient les nouveaux et anciens (nouvelles versions au besoin) services. Avec la seconde approche, les nouveaux services sont insérés dans l'AMS. Les nouvelles versions des services existants sont généralement insérés lorsque ces services-là n'exécutent pas ou après que leur exécution a été arrêtée. Cependant, plusieurs services (ceux de téléphonie notamment) fonctionnent en continu et ne peuvent être interrompus. Par conséquent, ils doivent être mis à jour pendant qu'ils sont en cours d'exécution.

Les deux approches ont des avantages et inconvénients. Quelques requis simples peuvent être dérivés pour analyser ces deux alternatives, dans le but de trouver une solution optimale par rapport à ces mêmes requis. Les requis sont les suivants :

1. L'interruption de service due à la mise à jour de l'agent devrait être minimal ;
2. Le délai entre une requête d'abonnement et la disponibilité pour l'utilisation d'un service devrait être minimale ;
3. La mise à jour ne devrait pas affecter le comportement des services. Plus clairement les modifications (personnalisations) effectuées par l'usager sur les services ne devraient pas être perdues ;
4. La solution devrait être adaptable à un nombre élevé de services. Autrement dit, la performance de la solution devrait être bonne même si l'usager s'abonne à 1000 services au lieu de 10 ;
5. La solution devrait être indépendante de la plate-forme d'agents mobiles utilisée pour son implémentation. Il y a plusieurs plates-formes disponibles sur le marché. La solution ne devrait exploiter les spécifités d'aucune plate-forme ;

6. La solution devrait être aussi simple que possible et ne devrait pas imposer de contraintes sur les architectures utilisées pour programmer les services. Plus spécifiquement, les developeurs ne devrait pas être obligés de développer leurs services suivant une méthodologie particulière pour que ceux-ci puissent être transportés et mis à jour dans un AMS.

Les requis 1, 3 et 4 sont les plus difficiles à combler avec une stratégie de remplacement de l'agent. Les services seront probablement interrompus durant la permutation de l'ancien AMS avec un nouvel AMS. Par ailleurs, la solution pourrait ne pas s'étendre à un nombre total (anciens + nouveaux) élevé de services puisqu'un nouvel AMS doit être reconstruit à partir de rien. Le délai pour la disponibilité des services pourrait devenir prohibitif si le nouvel AMS doit recharger aussi le code des anciens services (B et C dans ce cas ci). Finalement, les connaissances glanées par le vieil AMS sur les habitudes d'utilisation de l'usager, ainsi que les données ou règles relatives aux services que l'usager auraient explicitement modifiées dans le vieil AMS doivent être conservées.

Dans l'optique d'une stratégie de mise à jour dynamique de l'agent, l'insertion de nouveaux services est relativement facile à implémenter. C'est aussi le cas pour le remplacement d'un service quand ce service n'est pas en cours d'utilisation. Cette affirmation peut paraître surprenante car, malgré tout, l'insertion des services revient à fournir à l'AMS une intelligence qui lui permette de :

- charger les nouveaux services et leurs données quand il est informé qu'il doit transporter ces services pour l'usager ;
- être conscient en tout temps des services qu'il contient, de manière à mettre à jour son interface graphique quand il charge de nouveaux services ;
- utiliser les données pour manipuler les services et permettre la personnalisation de ces données.

Toutefois, les deux dernières exigences sont déjà implémentées et remplies pour tout AMS de notre architecture. Le supplément consiste donc simplement à transférer

(ouvrir des connexions "sockets") les codes exécutables et données des services, ce qu'un AMS qui transporte des pointeurs fait "routinement".

La solution semble donc évidente jusqu'à ce qu'on considère le cas où certains services sont mis à jour pendant qu'ils sont en cours d'exécution. A ce moment là, les requis 5 et 6 deviennent ardus à satisfaire. En effet, comment changer instantanément une application si on ne dispose pas d'information sur sa sémantique interne ? Pour assurer la continuité du programme, il faudrait peut être effectuer un transfert d'état des processus de l'ancienne version du service à la nouvelle version. Ceci implique l'utilisation de primitives du système d'exploitation, et par conséquent une certaine dépendance vis-à-vis de la plate-forme. Sinon, il faut probablement imposer un modèle architectural aux services qui permettrait de modifier certaines parties d'un service en cours d'utilisation.

4.1.2 Remplacement d'agent et mise à jour dynamique

D'un premier abord, la permutation de deux agents et la mise à jour dynamique d'un agent semblent similaires. En effet, remplacer un agent revient essentiellement à recharger les classes de cet agent et créer une nouvelle version. La plate-forme s'occupe ici du chargement et de l'initialisation. Toutefois, la plate-forme d'agents mobiles peut seulement charger un nouvel agent, pas insérer les classes d'un nouvel agent dans l'ancien agent, qui est plus sans interrompre l'exécution de cet agent.

La mise à jour dynamique ajoute du code où de l'intelligence à une application qui est en cours d'exécution sans interrompre ladite application. Plus clairement, les modifications dynamiques changent des parties de l'application. La difficulté principale avec ces changements "à chaud" est d'assurer l'intégrité du programme après le changement. Quels que soient les mécanismes utilisés, il faut préserver le programme des erreurs de définitions de type et ne pas briser les sémantiques du langage tels que l'association nom-objet. De nombreuses autres contraintes entrent aussi en ligne de compte selon les applications qu'on désire mettre à jour.

Avec le changement d'agent, nous essayons de permuter l'agent et les services qu'il contient avec un nouvel agent qui contient les anciens (éventuellement nouvelles versions) et les nouveaux services. Le problème est d'effectuer la permutation tout en remplissant les requis.

Les techniques de mise à jour dynamique introduisent des changements sur des éléments spécifiques d'un système. Par exemple, si une nouvelle version d'une classe ou d'un module est disponible, le système de mise à jour doit s'assurer que les appels de méthode sur les objets qui implémentent la vieille version sont plutôt redirigés vers des objets qui implémentent la nouvelle version. Bien entendu, ceci a le potentiel d'introduire des erreurs dans le programme. On peut considérer le cas simple où la nouvelle version de la classe ne supporte plus une méthode présente dans les versions antérieures de la même classe. Si cette méthode est invoquée vers la fin du programme alors que les objets ou classes ont été dynamiquement changés plus tôt dans le programme, ceci donnera lieu à l'invocation d'une méthode invalide (inexistante) dans un programme préalablement compilé. Conséquemment, les changements ne peuvent être introduits que sous certaines conditions.

La granularité des changements varie avec chaque système de mise à jour. Certains systèmes permettent des changements sur des entités aussi petites que des variables ou des fonctions (appeler la fonction provoquera l'appel automatique d'une autre). D'autres limitent les changements aux modules ou à des applications entières. Ces systèmes sont typiquement dépendants de la plate-forme car ils utilisent des primitives du système d'exploitation. On doit aussi recourir au système environnant (interprêteur ou système d'exploitation) pour transférer l'état entre applications. Les techniques de mise à jour qui ne nécessitent pas de support du système environnant imposent un modèle architectural aux applications pour qu'elles puissent être mises à jour.

4.2 Permutation d'agents

La permutation d'agents peut s'effectuer de deux manières : graduellement ou abruptement. Nous présentons ces deux approches. Le principal défi lors d'une

permutation d'agents est de s'assurer que les données des services sont correctement transmises. Ces données sont disponibles uniquement dans le vieil agent, car elles peuvent avoir été modifiées par l'usager ou par l'agent lui-même suite à l'étude des habitudes d'exécution de l'usager.

En effet, l'AMS offre une interface graphique où les services peuvent démarrer ou arrêter, mais aussi où l'usager peut entrer des règles d'exécution, i.e. tel service doit être démarré dès que tel autre est lancé, démarrer le service X avec les paramètres y, démarrer le service Z à telle heure, le service A utilise uniquement le service B, etc. Nous avons présenté uniquement les personnalisations que nous avons implémentées, il est bien sûr possible d'avoir des fonctionnalités beaucoup plus sophistiquées avec un produit commercial.

La permutation de deux agents peut se faire de deux façons : de manière abrupte ou de façon graduelle. Nous explicitons ces deux possibilités. Par la suite, nous présentons leur implémentation et la manière dont les problèmes tels que la synchronisation des données et les erreurs de transmission sont résolus.

4.2.1 Permutation graduelle et permutation abrupte

Que ce soit une permutation graduelle ou abrupte, l'UGS assemble d'abord un AMS qui contient des pointeurs aux exécutables des nouveaux et anciens services. L'AMS transfère ensuite les exécutables des UCS.

Dans le cas de la permutation graduelle, le nouvel AMS se déplace alors sur le site de l'ancien. Il recueille les données personnalisées au besoin et devient actif. L'ancien AMS devient inactif après avoir passé les données au nouvel AMS. Tout service ne pourra être maintenant démarré que par le nouvel AMS. Toutefois, les services qui étaient en cours d'exécution dans l'ancien AMS ne sont pas interrompus ; ils terminent avant que l'ancien AMS devienne totalement inactif.

Dans le cas de la permutation abrupte, le nouvel AMS ne se déplace pas sur le site où l'ancien AMS réside, après avoir copié les exécutables. Il transfère d'abord les données personnalisées (s'il y en a) du vieil AMS. Deux alternatives se présentent alors.

Dans la première, l'ancien AMS arrête les services qui étaient en cours d'exécution (au besoin), puis devient inactif. Dans la seconde, les services achèvent leur exécution avant que l'ancien AMS devienne inactif. Le nouvel AMS se déplace sur le site du vieil AMS aussitôt que celui-ci devient inactif. Il redémarre alors au besoin les services qui avaient été interrompues.

La permutation graduelle demande que deux AMS coexistent sur le même site durant la permutation. Ceci n'est pas toujours possible avec les petits moniteurs d'information (Palm Top, etc.). De là vient la principale motivation qui sous-tend la permutation abrupte. Bien qu'il soit théoriquement possible d'attendre que les services achèvent avant de réaliser la permutation, ceci n'est pas possible tout le temps. Un exemple est un service de valeurs boursières démarré par l'usager tôt dans la journée et qui doit rapporter l'évolution des actions sélectionnées toutes les heures.

Ces deux approches soulèvent les questions suivantes:

- Comment transférer les données et continuer à imposer des contraintes minimales sur l'AMS et ses services (interfaces, architecture des services, etc.) ?
- Avec la permutation graduelle, deux versions du même service peuvent s'exécuter concurremment durant la permutation. Comment maintenir la consistance entre les paramètres d'exécution de services qui exécutent concurremment ?
- Désactiver l'ancien AMS avant que le nouvel AMS se déplace sur le site expose la permutation abrupte aux pannes du réseau. Quels mécanismes de recouvrement sont prévus dans le cas où le déplacement du nouvel AMS échouerait ?

4.2.2 Implémentation des permutations

Les agents communiquent par le langage KQML que nous avons implémenté en utilisant la librairie JKQML de IBM alphaWorks. Ceci évite d'avoir à imposer des interfaces que les AMS implémenteraient pour communiquer par appel de méthodes.

Les AMS sont tenus d'implémenter le protocole d'échange des données illustré à la Figure 4.1. Nous n'avons pas utilisé d'ontologie et les utilitaires (tels JKP) présentés au chapitre 4 pour la permutation d'agent car ils introduisaient des erreurs difficilement

"traçables". L'expérience nous a aussi appris que, pour supporter l'ontologie, une machine d'inférence est quasiment indispensable. C'est un surplus de traitement très élevé que de demander à l'AMS de procéder à l'analyse de texte. Supporter un protocole est faisable s'il est très limité comme celui de la Figure 4.1. Sinon, il devrait être traité à l'extérieur de l'agent (librairies, utilitaires, etc.).

```
Service : Valeur_boursière
Parameters : Douala Wall_street Toronto
Start_Time : 9AM 8AM 8_30AM
End_Time : 5AM 9PM 6PM
```

Figure 4.1 : protocole d'échanges de données lors de la permutation

La figure 4.2 illustre la manière dont l'échange de données est implémenté dans les AMS.

```
Class MSA {
    Private void sendCustomization(...){
        String data = new String("Service : Valeur_boursière \n" +
                                 "Parameters : Douala Wall_street Toronto \n" +
                                 "Start_Time : 9AM 8AM 8_30AM \n" +
                                 "End_Time : 5AM 9PM 6PM \n\n");
        KQML request = new KQML();
        request.setPerformative(KQML.RECEIVE); //set KQML performatives
        request.setOntology(null);
        request.setContent(data); // subsequently send data
        //transmettre par socket
    }
}
```

Figure 4.2 : Échange de données sur les services durant la permutation

Permutation graduelle

Pour assurer la consistance des données dans les deux agents, l'utilisateur ne peut apporter de changements aux données relatives aux services après que la permutation eût été initiée. L'AMS n'offre donc plus aucune fonctionnalité dès que la permutation

graduelle commence. Il laisse tout simplement les services qui roulaient achever. Aucune modification ne peut être effectuée par l'usager.

Permutation abrupte

L'ancien AMS devient inactif avant que le nouvel AMS se déplace sur le site, mais il n'est pas détruit. Il s'enregistre tout simplement pour activation sur le disque dur. Il ne consomme ainsi plus de ressources systèmes. Un objet ou programme actif consomme des ressources. En effet, le processus d'activation (dans les architectures distribuées) permet d'enregistrer sur disque un objet qui est inactif pour une longue durée et qui utilise des ressources systèmes. L'objet (champs et méthodes) est alors enregistré pour le reste de sa période d'inactivité et ne consomme plus de ressources systèmes. Il est réactivé et chargé en mémoire à sa prochaine invocation. En bref, l'activation permet de préserver des ressources. Notre implémentation de l'activation est basée sur le modèle de l'objet réseau de Modula-3 NetObj.T.

Si le déplacement se déroule sans problème, le nouvel agent effacera l'ancien agent du disque dur à son arrivée. Si par contre un problème se produit lors du déplacement du nouvel agent, l'usager pourra toujours démarrer l'ancien agent car il est enregistré sur disque pour activation.

4.3 Mise à jour dynamique de l'AMS

Nous avons vu que l'investigation de l'ajout dynamique de services non en cours d'exécution était d'un intérêt mitigé. En effet, Il n'est même pas nécessaire de recueillir des données relatives à la performance de l'architecture dans ce cas là. Dans cette section, nous nous concentrons sur le cas où le service à mettre à jour est en cours d'exécution.

Tout d'abord, il faut clarifier ce que la mise à jour dynamique veut dire pour l'AMS. Mettre l'AMS à jour dynamiquement signifie changer l'implémentation d'un ou de plusieurs services pendant que ces services s'exécutent. Les services sont modifiés sans que l'application (l'AMS) ou le système ne soit interrompu et sans intervention

humaine. Les défis sont importants; toutefois, il existe déjà quelques implémentations et architectures de mise à jour dynamique. Notre solution par contre dépasse et améliore toutes les réalisations précédentes par sa modularité et sa facilité d'utilisation, ainsi que la précision et le contrôle qui peut être exercé sur les changements. Elle est facilement utilisable et n'occupe que 9 Kilo-octets d'espace mémoire. Les fournisseurs de services peuvent activement sélectionner les parties (objets) du système qui devraient être modifiées et spécifier une politique de changement pour chaque version antérieure. Plusieurs versions de tout module dynamique peuvent s'exécuter concurremment dans le programme. Les mises à jour dynamiques sont transparentes à la programmation et à l'exécution des services. L'AMS contrôle la version de chaque partie du programme.

Les différentes architectures et implémentations de mise à jour dynamique varient grandement selon les paramètres suivants : granularité des changements, support du système d'exploitation ou de la machine virtuelle, intervention humaine et modèle de programmation des applications dynamiques. L'unité de changement peut être aussi petite qu'une fonction ou variable, ou aussi grande qu'une librairie ou programme entier. Dans notre implémentation, les changements peuvent être introduits uniquement sur des classes. Plus clairement, toute nouvelle version d'une classe dynamique peut être chargée en mémoire et utilisée pour les créations subséquentes d'objets dynamiques. L'usager sélectionne aussi les instances existantes qu'il désire remplacer par des instances de la plus récente version de la classe. Cette unité de changement est consistante avec le paradigme orienté-objet et est implémentable en Java grâce à ses mécanismes d'édition dynamique de liens. Les appels sur chacune des instances remplacées seront redirigés selon la politique de conversion de la version de cette instance, si elle existe.

L'implémentation utilise l'interface *Java Core Reflection* de JDK 1.3 (Java Development Kit). Elle est basée sur la redirection des appels grâce à un proxy et requiert l'utilisation de librairies du MSA qui permettent les changements dynamiques. Les évaluations expérimentales indiquent que cette approche est meilleure par rapport aux requis concernant l'interruption des services, le délai pour la disponibilité des services et l'adaptabilité pour un grand nombre de services.

4.3.1 Concepts de base et préalables

La première notion qu'il faut saisir est celle de la résolution des liens inter-modules d'un programme par un éditeur dynamique de liens. Une application est composée de modules, chaque module est une collection de procédures, variables et constantes. Un module peut exporter certaines de ces caractéristiques dans une interface et peut importer d'autres modules. Durant la compilation, le compilateur s'assure que toute invocation d'un élément importé coïncide avec la déclaration de cet élément dans son fichier source. À la différence des éditeurs de liens statiques, les éditeurs de liens dynamiques ne résolvent pas les références établis par importation/exportation à la compilation. Ils le font plutôt plus tard à un moment dédié que nous appellerons le temps d'éditions des liens (Franz, 1997). Toutefois, dans la plupart des implémentations d'édition dynamique, le temps d'édition des liens coïncide souvent avec le moment où le code exécutable du module est chargé en mémoire. Ceci est le cas avec Java qui est un "lazy linker", i.e. résolution des liens seulement quand le module importé est invoqué pour la première fois durant l'exécution du code. Il existe plusieurs variations d'édition de liens dynamiques. Nous illustrons le cas de Java à la Figure 4.3.

Au moment de l'édition des liens, la machine virtuelle Java (Java Virtual Machine JVM) parcourt le fichier objet (.class) de la classe A et son réservoir de références ("constant pool"), et résout les références aux classes externes. Cette édition des liens se fait la première fois que les classes externes (importées) sont rencontrées dans le programme. Ainsi, l'édition des liens et le chargement des fichiers se fait une classe à la fois.

À la Figure 4.3, la première fois que la JVM traite l'instruction "B b = new B()", il examine le réservoir de référence de la classe A, trouve la structure appropriée et se déplace à l'index indiqué (38 sur la figure) pour récupérer de l'information sur la classe B. Dans l'exemple, l'information est le nom absolu de la classe en représentation unicode.

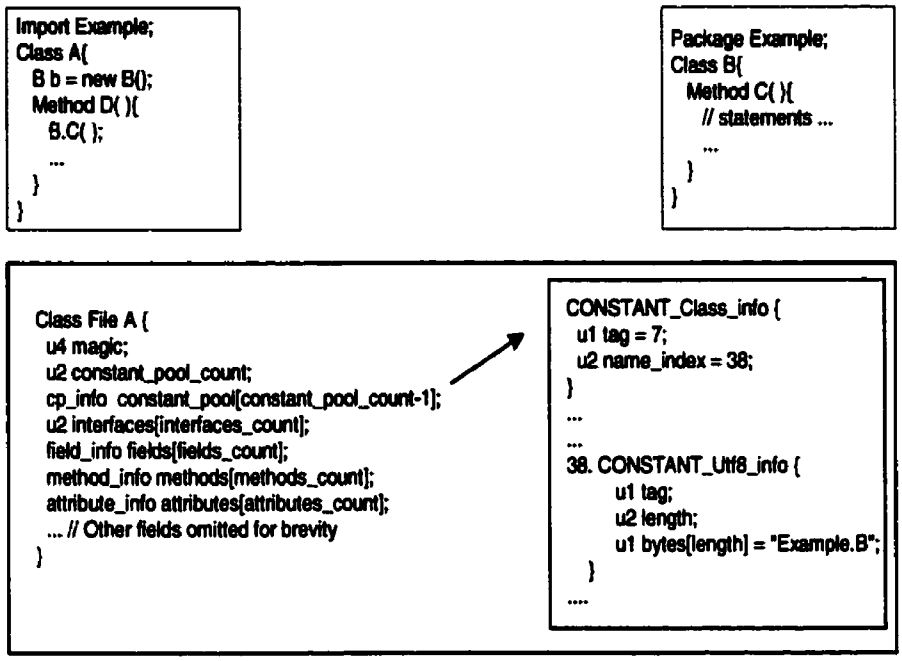


Figure 4.3 Modules et code objet Java

Ce processus est réalisé une seule fois par classe par programme. Ceci est dû au fait qu'une fois qu'une classe comme la classe B est résolue, la JVM maintient un lien inamovible à cette représentation de la classe. Conséquemment toutes les invocations futures de la classe utiliseront ce lien sans la pénalité d'avoir à fouiller le code objet. Ceci veut aussi dire qu'une classe ne peut pas être changée dynamiquement une fois qu'elle a été résolue, à moins que le système de mise à jour n'accède le monceau (heap) de la JVM et change ce lien pour qu'il pointe à une autre version de la classe. Pour cela, il faut carrément changer la JVM, ce qui implique que la solution n'est plus portable et introduit d'autres difficultés que nous aborderons plus tard. Toutefois, différentes versions de la même classe peuvent être chargées tant que la classe n'est pas résolue. La classe peut toujours être manipulée à travers la superclasse "Object" qui est la superclasse de toutes les classes écrites en Java.

La question qui se pose maintenant est la suivante : comment garantir l'intégrité du programme après le changement puisque l'AMS n'a pas accès à la pile du processus ?

Les mises à jour "à chaud" peuvent potentiellement causer des erreurs de violation de type. Bien que l'intégrité des types en Java n'ait pas encore été formellement prouvée (Drossopoulou *et al.*, 2000), en pratique le vérificateur de bytecodes de Java prévient les erreurs de violation de type, excepté dans certains cas pathologiques (Freund *et al.*, 1998). Dans le but d'éviter des erreurs de violation de type, les systèmes de mise à jour dynamiques introduisent généralement des changements après s'être assurer (mécanismes de verrouillage et synchronisation, vérification de la pile du programme) que la cible (classe, procédure) n'est en utilisation nulle part ailleurs dans le programme au moment du changement. Ces techniques sont généralement dépendantes de la machine puisqu'elles utilisent des primitives du système d'exploitation et effectuent des appels à son noyau (kernel).

Qui plus est, la plupart de ces implémentations exige l'assistance humaine pendant la mise à jour et demande que le developeur soit ré-entraîné de manière à pouvoir programmer en fonction du système de mise à jour. Une erreur de violation de type est illustrée à la Figure 4.4.

La classe Service_i crée une instance de Dependent_i ("di") correspondant à la première version située à gauche sur la figure. Toutefois, avant une des invocations de l'objet "di", il est dynamiquement changé à la nouvelle version qui n'implémente plus la méthode Z(). Lorsque Service_i essaie d'appeler Z, une violation de type se produit.

Les violations de type se produisent aussi quand un objet est invoqué par l'intermédiaire d'une interface incompatible. Imaginez le cas où la première version de Dependent_i implémente une interface quelconque que nous appellerons Interface_i. Une instance de Dependent_i peut être "castée" et manipuler à travers cette interface. Toutefois, dans le cas où la seconde version ne supporterait plus l'interface, il y aura une erreur de violation de type la prochaine fois que l'instance "castée" sera manipulée si l'objet auquel elle réfère a été dynamiquement changé pour refléter la nouvelle version.

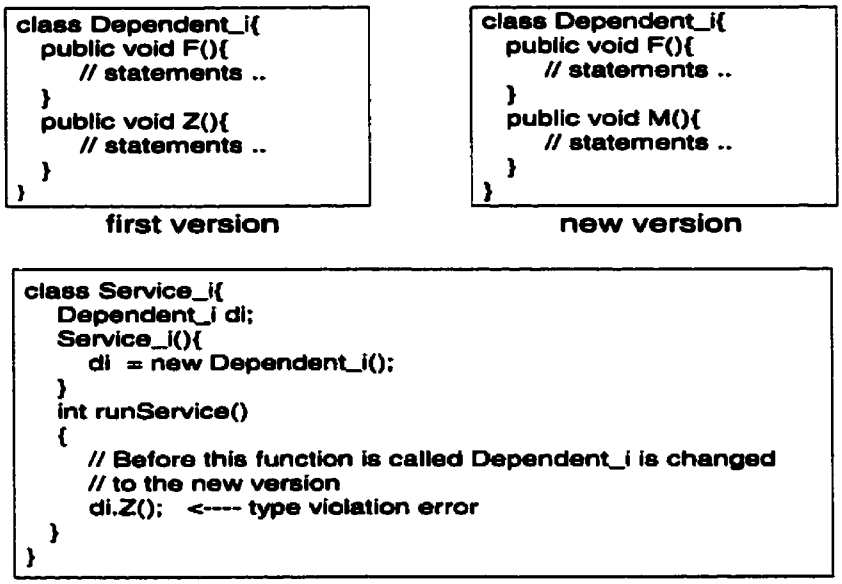


Figure 4.4 Violation de type causée par un changement dynamique

4.3.2 Approches de mise à jour

L'AMS offre les mises à jour dynamiques à travers les classes dynamiques dont l'implémentation peut être changée pendant l'exécution du service. Les mises à jour de classes dynamiques peuvent être appliquées selon différents modèles.

Les changements peuvent affecter toutes, quelques-unes unes ou aucune des instances de la classe qui a été modifiée. Selon la méthodologie de mise à jour, différentes questions se posent sur le traitement des classes dépendantes, la coexistence de plusieurs versions de la même classe dans le même programme, etc. Nous décrivons les alternatives et présentons la méthodologie de mise à jour que nous avons implémentée. Tout le long du texte un objet dynamique réfère à une instance d'une classe dynamique.

Sémantiques de mise à jour

Les sémantiques de changements de version sont documentées (Gray *et al.* 1997, Malabarba *et al.* 2000). Nous les analysons brièvement. Une première approche pour mettre à jour les objets existants d'une classe qui a été modifiée est de bloquer la création de nouveaux objets de la classe jusqu'à ce que toutes les vieilles instances expirent. Des objets de la classe pourront de nouveau être créés à la fin de ce délai. Ceci peut prendre un temps arbitrairement long.

Une autre approche est de changer toutes les instances existantes de la classe au moment du changement pour qu'elles reflètent la nouvelle version. Ici, des mécanismes de verrouillage permettent de s'assurer qu'aucune instance de la classe n'est utilisée au moment du changement. Les codes objets des autres classes sont parcourus pour s'assurer qu'ils ne contiennent pas de références à des méthodes ou champs obsolètes de la classe modifiée. Les instances des sous-classes sont aussi modifiées (débarrassées de champs et méthodes obsolètes). À moins que des mécanismes de transfert d'état soient en place, l'état interne des anciens objets ne peut être copié dans les objets nouvellement créés. Les éventuels mécanismes de transfert d'états sont rigides et sont possibles uniquement pour des architectures spécifiques et bien définies qui obligent les classes à supporter certaines interfaces non modifiables. Conséquemment, le transfert de contexte est limité à des applications restreintes. Une autre conséquence est que les programmes doivent être écrits en fonction des mises à jour dynamiques.

Face à ces difficultés, la plupart des systèmes qui implémentent cette approche créent tout simplement de nouveaux objets vides qui remplacent les anciens objets. Bien entendu, il y a l'effet néfaste que toutes les valeurs (souvent importantes) contenues dans les champs de ces anciens objets sont perdues. Cette perte d'information se propage selon une réaction en chaîne, i.e. les données et traitements contenus dans tout objet dont les uniques références sont ainsi purgées, sont aussi perdues. En définitive, bien que les changements puissent être configurés de sorte qu'il puisse être prouvé que le programme sera toujours syntaxiquement correct et l'intégrité des types préservée, il y a une discontinuité de fonctionnement dans le comportement et l'exécution du programme. Le

fardeau retombe sur les épaules du programmeur qui doit alors parvenir à s'assurer d'une façon quelconque que le programme continuera à produire des résultats valides. Pas des résultats corrects tout simplement parce que le programme retourne un entier comme il était supposé, mais que la valeur de cet entier soit celle que le programme était supposé produire s'il disposait de l'information perdue. Cette pénalité réduit considérablement le coté pratique de la méthode.

Une troisième alternative est de laisser les anciennes instances des classes dynamiques inchangées au moment de la mise à jour. Toutefois, tous les objets créés à partir de ce moment là refléteront la nouvelle version. Ce schéma est basé sur l'idée qu'une fois que les vieilles instances seront éliminées par le ramasse-miettes, seule la nouvelle version sera en fonctionnement. N'empêche, plusieurs versions de la même classe coexistent dans le même programme sans que le developpeur n'ait de contrôle sur la version d'un objet particulier. Par conséquent, une ambiguïté nuisible règne toujours à propos de quelle version de la classe est utilisée lors d'une invocation. Le résultat final est que le comportement du système est imprévisible. Cette approche viole donc les sémantiques d'association nom-classe. Ici aussi, la solution à court terme est de concevoir son architecture de manière à pouvoir se repérer dans son programme et profiter des mises à jour dynamiques.

Avec toutes ces approches, le système de mise à jour impose des contraintes sur le design des applications et ainsi influence ou pire décide du comportement de l'application. Normalement, les applications devraient décider de leur politique de mise à jour et utiliser les mises à jour dynamiques à leur guise dans le but de fonctionner sur une longue durée sans interruption. Autrement dit, l'utilisation du système de mise à jour devrait être adaptable à l'application et cela ne devrait pas être l'application qui s'adapte au système de mise à jour.

Modèle de mise à jour de l'AMS

L'AMS permet aux developeurs de choisir précisément quels objets dynamiques doivent être mis à jour. Il leur permet aussi de décider de la politique de mise à jour qui

doit être appliquée d'une version d'une classe à une autre. Clairement, chaque application décide de sa politique de mise à jour et de l'étendue des mises à jour. Les parties du programme qui ne sont pas modifiées sont laissées à leurs versions respectives. Les programmeurs peuvent ainsi modifier des régions de l'application sans qu'il n'y ait d'impact sur les autres parties. Ils peuvent s'assurer de la continuité de fonctionnement du programme en ciblant les portions de code qui contiennent des défauts ou qui nécessitent des améliorations pour modification et en laissant les objets qui contiennent des données critiques pour l'application inchangée. Le système (AMS) fournit des méthodes qui permettent de connaître la version de tout objet. Une clé unique permet d'identifier chaque objet. Cette clé est composée de la classe de l'objet dynamique, sa version, son chargeur de classes, et une clé fournie par l'usager. Les classes sont effectivement redéfinies à la granularité d'objets.

Les objets dynamiques sont manipulés uniquement à travers des "proxies" dynamiques de classes. Un proxy dynamique d'une classe défini par l'AMS supporte les interfaces de la version de la classe qui était active (la plus récente à ce moment-là) au moment où ce "proxy" a été créé. Toutefois, les objets dynamiques peuvent être changés pour refléter la plus récente version sans que cette version (la plus récente) ne soit tenue d'implémenter aucune des interfaces des versions antérieures. Les appels aux méthodes obsolètes sont interceptés au proxy et traités selon les instructions (spécifiées dans la politique de mise à jour) du developpeur du service. Les classes dynamiques sont manipulées uniquement à travers les interfaces de leurs proxies. Cette légère restriction assure que l'intégrité des types est préservée.

Dans notre implémentation, les classes dynamiques ne sont jamais résolues, toutefois leurs interfaces le sont. Chaque nouvelle version d'une classe dynamique est chargée à travers un autre chargeur de classes. L'AMS coordonne l'utilisation des classes dynamiques. La Figure 4.5 illustre le modèle de mise à jour de l'AMS.

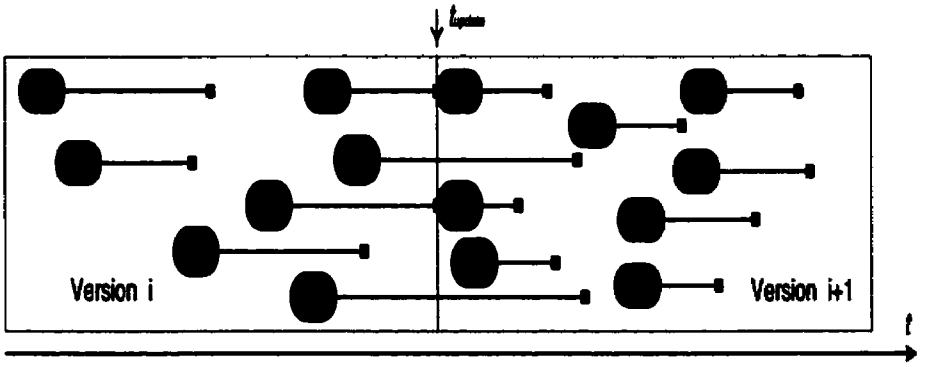


Figure 4.5 Modèle de mise à jour dynamique de l'AMS

Les objets sont d'abord créés en utilisant la version i et sont manipulés à travers l'interface i1. À partir de l'instant de la mise à jour (t_{update}), les objets dynamiques sont créés en utilisant la version i+1. Certains objets de la version i sont remplacés par des objets de la version i+1. Il est toujours possible de manipuler ces objets qui ont été changés à travers l'interface i1, même si la nouvelle version (i+1) n'implémente pas cette interface. L'AMS se charge du transfert d'appel des méthodes qui ont les mêmes paramètres. Toutefois, les objets nouvellement créés (version i+1) ne peuvent être accédés que par l'interface i2.

4.3.3 Implémentation de l'évolution dynamique

Cette solution classifie les objets dynamiques selon la version de leur classe et leur clé unique fournie à l'initialisation. Bien entendu, les méthodes statiques de la classe opèrent sur les objets qui sont de la même version de cette classe uniquement. L'implémentation est basée sur les proxies dynamiques de JDK1.3. L'interface proxy (de l'ensemble Reflection) permet au developeurs de créer des proxies de classes consistant en un ensemble quelconque d'interfaces. Chaque objet proxy a un objet associé ("Invocation handler") où toutes les invocations de méthodes seront envoyées. La Figure 4.6 lance un bref coup d'œil sur la classe Proxy de Java 1.3

```
Public class Proxy {
    static object newProxyInstance(ClassLoader loader, class[] interfaces, InvocationHandler h);
    static InvocationHandler getInvocationHandler(Object proxy);
    // ... other methods omitted
}
```

Figure 4.6 Classe java.lang.reflect.Proxy de JDK 1.3

La classe "MSAFactory" est l'interface par laquelle les objets dynamiques sont créés et mis à jour. Il y a un MSAFactory par AMS. Une clé doit être fournie pour la création de tout objet dynamique. La clé est une chaîne de caractères quelconque et doit être unique pour la classe de l'objet dynamique. Le descripteur de type d'un objet dynamique est composé de sa classe, sa version, son chargeur de classes et de la clé fournie à la création. Il est important de conserver les clés utilisées pour créer les objets dynamiques. Les programmeurs les utiliseront plus tard pour indiquer quels objets dynamiques doivent être mis à jour pour refléter la dernière version. La classe MSAFactory maintient un lien entre chaque classe et son gestionnaire des mises à jour "MSAClassManager". L'AMS charge la classe MSAFactory sur tout hôte où il se déplace. La Figure 4.7 présente des extraits de la classe MSAFactory.

Une classe est mise à jour en appelant "setClassVersion" de MSAFactory. Cette nouvelle version sera utilisée pour la création de toutes les prochaines instances. Le dernier paramètre de createInstance est un tableau contenant les arguments pour invoquer le constructeur de "ClassName". Celui de "updateInstance" est un tableau bi-dimensionnel d'arguments pour les constructeurs des objets qui doivent remplacer les objets mis à jour. Le second argument "keys" contient la liste des clés indiquant les objets qui doivent être modifiés. Les éléments situés aux mêmes positions d'index dans "keys" et "arguments" sont utilisés lors du remplacement des objets dynamiques. Omettre d'appeler "updateInstance" implique qu'aucun objet existant ne doit être changé. Seules les prochaines instantiations refléteront la nouvelle version.

```
public final class MSAFactory{
    static Object createInstance(String ClassName, String instance_key_id,
                                 Object[] arguments) throws InvocationTargetException {
        m = addClassManager (ClassName);
        // instantiates a class manager for the class ClassName if it is
        // its first instance that is being created maps ClassName and its classManager and returns it
        return (m.getProxyObject (instance_key_id, arguments) );
    }
    static void updateInstance(String ClassName, String[] keys,Object[][] arguments) {
        m = getClassManager(ClassName);
        m.updateObjects(keys, arguments);
    }
    static void setClassVersion(String classname, String version, URL classLocation,){
        m = getClassManager(classname);
        m.setVersionLocation(version, classLocation);
    }
    static void setUpdatePolicy (String classname, String[] policy){
        m.setRedirectPolicy (policy);
    }
    // other protected or private coordination fields and methods omitted for brevity
}
```

Figure 4.7 La classe MSAFactory implémente l'interface des classes dynamiques

Les clés fournies pour une mise à jour doivent être choisies parmi celles qui ont été utilisées dans des appels à "createInstance". Un objet dynamique est créé uniquement à travers "createInstance". Cet objet dynamique est un proxy qui supporte les interfaces de la dernière version de la classe nommée "ClassName" (premier paramètre de createInstance). La fonction "setUpdatePolicy" permet de définir la politique de redirection des invocations de méthodes de certaines versions précises à la version actuelle de la classe "classname".

Une politique de redirection des appels peut être spécifiée une seule et unique fois d'une version à une autre. Les mises à jour se font uniquement dans un sens : d'une version antérieure à la plus récente version de la classe. L'on ne peut pas modifier, mettre

à jour un objet qui reflète déjà la dernière version de la classe. Les politiques de redirection des appels d'une version à une autre sont très simples à écrire. La première ligne indique la version antérieure concernée, chacune des lignes suivantes contient deux noms. Le premier est celui de la méthode dans l'ancienne version suivie du nom de celle qu'il faut appeler à la place dans la nouvelle version. Un appel ne peut être redirigé d'une fonction à une autre que si les deux fonctions acceptent les mêmes arguments. La politique concernant une version est conservée dans une chaîne de caractères. Un vecteur de chaîne de caractères permet de spécifier des politiques de redirection pour plusieurs versions à la fois. Une politique de redirection simple ressemble à celle sur la Figure 4.8.

```
i.i  // Class Version
A A1 // Invocation Redirection From A To A1
B B1 // Invocation Redirection From A To A1
```

Figure 4.8 Exemple de redirection des invocations pour la version i.i d'une classe quelconque

La première ligne indique la version de la classe à laquelle cette politique de redirection s'applique. Les lignes suivantes indiquent les redirections : la méthode A1 de la nouvelle version doit être appelée en lieu et place de la méthode A, la même chose pour B1 et B. Une méthode peut être redirigée uniquement vers une autre méthode qui accepte les mêmes paramètres (arguments). Les invocations subséquentes sur n'importe quel objet dynamiquement modifié seront traitées selon la politique de mise à jour que le fournisseur de service ou developpeur a ordonnée. Si aucune politique n'a été indiquée et que des objets dynamiques sont mis à jour, les invocations seront redirigées vers la méthode du même nom acceptant les mêmes paramètres qui sera trouvée dans la nouvelle version. L'AMS ne tient pas compte de la valeur de retour (void, int, etc.) ou du

modificateur (static, public, etc.) de la méthode trouvée. Bref, il ne vérifie pas sa signature.

Exemple d'utilisation

Un exemple de mise à jour dynamique utilisant l'AMS est présenté à la Figure 4.9. Les services utilisent MSAFactory (createInstance) pour créer leurs objets dynamiques. La librairie de mise à jour dynamique sera mise à la disposition des developpeurs de services. Ils peuvent donc simuler le comportement de leur service avant, pendant et après une modification dynamique. La méthode updateInstance est utilisée pour changer des objets précis.

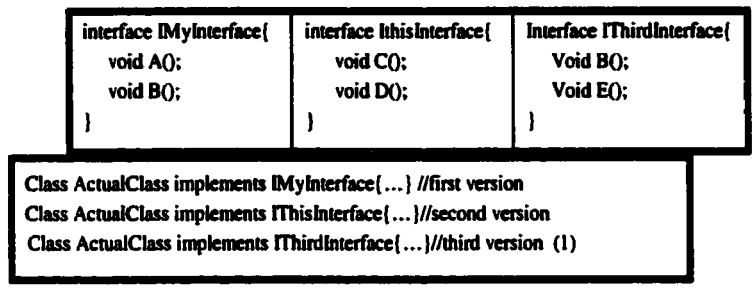
La figure présente trois exemples de création d'objets dynamiques, deux mises à jour de deux des objets, l'entrée d'une politique de redirection et les manipulations qui peuvent être faites sur les objets à la suite de ces changements. Elle illustre le comportement par défaut qui stipule que : quand aucune politique de redirection n'est spécifiée, les invocations sur les instances dynamiques changées sont redirigées vers une fonction du même nom ayant les mêmes arguments dans la nouvelle version. Bien entendu, si aucune fonction du même nom n'est trouvée, une exception est levée.

Il est facile de constater que fournir des méthodes pour identifier et modifier (au cas par cas au besoin) des parties d'un service est la façon la plus flexible et la plus pratique de supporter des mises à jour dynamiques. C'est aussi l'approche qui donne le plus de contrôle sur l'application et les mises à jour dynamiques au programmeur.

4.3.4 Gestion des classes et acheminement des invocations

Les interfaces retournées par MSAFactory sont des proxies dynamiques créés par l'intermédiaire de la classe Proxy de Reflection. Les objets "MSAClassManager" coordonnent les proxies. Il y a un objet MSAClassManager par classe dynamique. Une vue de la classe "MSAClassManager" est présentée à la figure 4.10. Deux instances de classes dynamiques différentes peuvent avoir la même clé. Toutefois deux instances de la même classe doivent avoir des clés différentes. Il y a un objet "InvocationHandler" par

objet dynamique. Chaque objet "MSAClassManager" coordonne tous les "InvocationHandler" de toutes les instances de sa classe.



```
void doSomething(...){
        MSA.startFactory();
        MSAFactory.setClassVersion("ActualClass", "1.0", "142.133.xx.xxx");
        IMyInterface im1 = (IMyInterface) MSAFactory.createInstance(
                        ActualClass.class.getName(),"keyString1",new Object[]{...});
        IMyInterface im2 = (IMyInterface)  MSAFactory.createInstance(
                        ActualClass.class.getName(), "keyString2",new Object[]{...}); (2)
        // method code ...
        MSAFactory.setClassVersion("ActualClass", "2.0", "Arbitrary location");
        // Method A is redirected and method B is not supported anymore
        MSAFactory.setUpdatePolicy("ActualClass",
                        new String[]{"1.0 \n  A  C \n"});                    (3)
        // im1 is explicitly upgraded.
        MSAFactory.updateInstance(ActualClass.class.forName(),
                        new String[]{"keyString1"}, new   Object[][]{...});
        // this will invoke method C on the new internally created ActualClass object see (3)
        im1.A();
        // a NoSuchMethodException will be thrown since the policy ignored this method
        im1.B();
        // No changes with both of these instructions. the object created at (2) wasn't changed
        im2.A();
        im2.B();
        // Of course it also possible to execute the line below
          IThisInterface it1 = (IThisInterface) MSAFactory.createInstance(
                        ActualClass.class.forName(),"keyString3", new Object[]{...});
        // later on
        MSAFactory.setClassVersion("ActualClass", "3.0", "Wherever");
        // update im2
        MSAFactory.updateInstance(ActualClass.class.forName(),
                        new String[]{"keyString2"}, new   Object[][]{...});
        // No update policy specified.
        // This will invoke A on ActualClass Version 3 at (1) if ActualClass version 3 implements A
        // even though A is not declared in any interface of  Actual Class version 3.
        // If ActualClass does not implements A, a NoSuchMethodException is thrown
        im2.A();
        // This will work calling method B() declared in IthirdInterface
        im2.B();
}
```

Figure 4.9 Exemple d'utilisation des librairies de l'AMS pour une mise à jour dynamique

Les méthodes statiques ne s'appliquent qu'aux objets qui sont de la même version de la classe. Cette approche est la façon la plus pratique d'implémenter les classes dynamiques. Sinon, il faudrait imposer que les différentes versions d'une même classe supporte une fonction de transfert, ou encore exiger que les méthodes statiques soient

supportées par toutes les versions de la même classe. Il n'est pas nécessaire d'interdire les méthodes statiques.

Il serait possible d'avoir plus de généralité et permettre le transfert des invocations d'une méthode à n'importe quelle autre peu, importe ces arguments. Toutefois, ceci devrait allonger considérablement le temps d'exécution à cause des vérifications *et al*locations de mémoire supplémentaires (réallouer un nouveau tableau d'objets pour contenir les arguments de la fonction "invoke" de *Reflection*).

Avec une édition des liens dynamiques, il est impossible de changer une référence à une classe ou interface déjà résolue. C'est pourquoi les interfaces qui sont utilisées dans le programme avant la mise à jour ne peuvent être changées. Toutefois, tant et aussi longtemps que la référence n'est pas résolue, la même classe ou interface peut être rechargée autant de fois que nécessaire. Un nouveau chargeur de classes ("Classloader") est instancié chaque fois qu'une classe doit être rechargée.

```
Class MSAClassManager{
        String sClassName, sClassLocation, sCurrentVersion;
        // Other fields ommitted
        public Object getProxyObject( String key, Object[] args){
                MSAClassLoader ms = new MSAClassLoader(sClassLocation, sClassName);
                Class c = ms.findClass(); Class ca[] = new Class[args.length];
                for(int i=0;  i<args.length;i++)  ca[i] = args[i].getClass();
                 Object o = c.getConstructor(ca).newInstance(args);
                // add the method invocation forwarder to the list of objects of this update manager and
                // create a internal map with a unique key
                MSAInvocationForwarding mif = new MSAInvocationForwarding(o);
                setTypeDescriptor(ms,c, mif, sCurrentVersion, key);
                setVersion(mif, sCurrentVersion);
                return(MSAProxy.newProxyInstance(ms, o.getInterfaces(),mif));
        }
        public void setRedirectPolicy(String[] policy) {
                // Parse the update policy array. Each index in the array refers to a different version
        }
        public String getRedirectPolicy(String version){
                // Returns the redirect policy from the version "version" to the current version
        }
        public void updateObjects(String[] key, Object[] args){
                // For each key
                MSAInvocationForwarding mif = getInstanceHandler(key[i]);
                MSAClassLoader ms = new MSAClassLoader(sClassLocation, sClassName);
                Class c = ms.findClass();
                Object o = c.getConstructor(new Class[]{c}).newInstance(args);
                mif.setRedirect(getRedirectPolicy(getVersion(mif)));
                mif.replaceObject(o);
        }
        private void setTypeDescriptor(ClassLoader cl, Class c,
                  MSAInvocationForwarding m, String version, String key) {
                // uniquely identifies each instances of this class
        }
        // returns the version of the object currently handled by the MSAInvocationForwarding object
        private String getVersion(MSAInvocationForwarding mif){...}
         // set the version of the object handled by the "mif" object
        private void setVersion(MSAInvocationForwarding mif, String sVersion);
        // returns an array of the interfaces supported by the ten previous versions of the class
        // ...others method omitted for brevity
}
```

Figure 4.10 Gestion des classes dynamiques

```
Public class MSAInvocationForwarding implements InvocationHandler{
    Object actual_object;          RedirectTable redirectTable;
    public MSAInvocationForwarding(Object o){
            actual_object = o;
    }
    public Object invoke(Object proxy, Method method, Object[] args)  throws InvocationTargetException,
            NoSuchMethodException, SecurityException,  IllegalAccessException, IllegalArgumentException  {
            Object o = null;
            Try {
                    o = bRedirect ? ((Method)RedirectTable.get(method)).invoke(actual_object, args) :
                        method.invoke(actual_object,args);
            }catch (Exception ex){
                    o = getLookup(method,args).invoke(actual_object, args);
            }
            return o;
    }
    public synchronized void setRedirect(String redirectPolicy){
            redirectTable.setTable(redirectPollicy) // set the redirection table "redirectTable"
    }
    public synchronized void replaceObject(Object o){
            // Scans o1 and o2 methods through reflection
            redirectTable.mapMethodRedirection( actualObject, o);
            //and saves every method pair (redirection) in its table following the redirectPolicy
            actual_object = o;
    }
    Method getLookup(Method m, Object[] args){
      // look up the method and return it when it is first accessed on actual object
     // algorithm enables log(n) order performance N being the number of redirected methods for this class.
    }
  // other coordination methods
}
```

Figure 4.11 Acheminement des invocations aux classes dynamiques

4.4 Évaluation sommaire des schémas de mise à jour

L'ajout, le retrait et la maintenance des services sont les premières préoccupations à satisfaire pour toute architecture de services réaliste. En effet, le déploiement rapide de nouveaux services innovateurs est primordial pour le succès des fournisseurs de services. La performance de leurs applications est un autre important facteur de démarcation. Or, nous n'avons pas abordé la performance ni même évalué les solutions par rapport aux requis dans ce chapitre. Pour remédier à ce manque, le prochain chapitre traitera de la performance de l'architecture. Il procédera aussi à l'évaluation des stratégies de mise à

jour présentées ici. Nous aurons alors assemblé les deux éléments fondamentaux qui sont indispensables pour le développement futur de l'abstraction pour la mobilité des services qu'est l'AMS.

Pour un programmeur habile de systèmes d'agents, la permutation de deux agents où la mise à jour dynamique d'un agent avec des services qui ne sont pas en cours d'exécution est relativement simple à réaliser. À l'opposé, la mise à jour dynamique d'un service en cours d'exécution est plus difficile.

La stratégie de la permutation d'agent à deux variantes. La première, la permutation graduelle, ne cause pas d'interruption de service mais est difficilement applicable pour les services qui s'exécutent continuellement ou prennent un temps très élevé avant de terminer. La seconde, la permutation abrupte, cause une (brève) interruption de service.

La mise à jour dynamique de l'agent est excitante parce qu'elle dépasse et étend les limites de notre recherche car elle est applicable à n'importe quel programme informatique écrit en Java. Si on se confine au problème de la mise à jour de l'AMS, elle est toujours plus intéressante que la permutation, car il n'est pas nécessaire de créer un nouvel agent et le délai pour la disponibilité des services est réduit (par rapport à la permutation des agents).

L'ambition de *mettre à jour les services qui s'exécutent continuellement sans les interrompre* nous a amené à développer une solution générique pour la mise à jour sans interruption des logiciels Notre principale exigence lorsque nous implémentions le mécanisme était que les changements dynamiques soient aussi simples que possibles. L'objectif de généricité a été atteint, au point où (presque par inadvertance) notre librairie se révèle être une contribution majeure dans le domaine de l'évolution dynamique des programmes informatiques. C'est la première approche documentée qui combine la sélectivité des changements, la possibilité d'adapter ces changements pour chaque version, la réduction de la taille et la non nécessité de support système. Le mécanisme de base pour l'évolution des versions est l'ajout d'interfaces. L'usager ajoute ou change les interfaces des classes qu'il veut modifier dynamiquement. Ces nouvelles interfaces seront

utilisées pour les nouvelles instances et une politique de redirection peut être spécifiée pour rediriger les appels effectués sur les anciennes interfaces sur les nouveaux objets.

L'évolution dynamique des programmes est un domaine actif de recherche. Toutefois, c'est dans le souci de trouver une solution à un problème qui affecte les services téléphoniques que nous avons (les premiers) utilisés Java pour montrer que l'évolution dynamique était possible à un haut niveau avec une petite librairie de 9 Kilo-octets.

CHAPITRE V

ÉVALUATION DE PERFORMANCE

Nous avons présenté une nouvelle architecture pour la mobilité des services au chapitre 3. Le chapitre 4 a exposé le problème de la mise à jour de l'agent mobile pour services (AMS) dans cette architecture, lors de nouvelles souscriptions. Il a aussi exposé l'implémentation de deux stratégies possibles pour effectuer cette mise à jour: la permutation d'agents et la mise à jour dynamique d'un AMS. Nous commençons ce chapitre en procédant à une analyse de performance de l'architecture. Cette analyse permettra de décider si l'architecture est viable. Nous continuons avec une évaluation des requis énoncés au chapitre 4 pour la technique de permutation d'agents. Finalement, nous examinons la performance de la mise à jour dynamique et son comportement en regard des requis du chapitre 3.

5.1 Analyse de performance du paradigme

Notre évaluation de performance déterminera le temps nécessaire pour assembler et lancer un AMS ainsi que la performance d'une application exécutée par l'intermédiaire de l'AMS. Pour débuter, nous séparons les diverses sources de délais du paradigme AMS et procédons à une brève revue des analyses de performance trouvées dans la littérature. Nous continuons en présentant notre modèle d'échantillonnage et les conditions sous lesquelles nous avons prélevé nos mesures. Finalement, nous exposons nos résultats.

5.1.1 Analyse du paradigme agent

Il y a deux points d'intérêt à mesurer pour notre architecture : le temps écoulé entre une souscription et la disponibilité (pour exécution) des services sur la machine de l'usager (1) ainsi que l'impact de performance causé par l'AMS sur l'exécution d'un service donné (2).

1. Notre architecture impose un délai parce que les services sont déplacés pour être exécutés sur la machine de l'usager. Par contre, avec une architecture client-serveur, il serait possible d'utiliser les services immédiatement après l'abonnement. Il est donc primordial que le délai pour la disponibilité des services ne soit pas trop grand dans l'éventualité où les usagers désireraient profiter des services immédiatement.
2. L'AMS a un impact sur l'exécution des services qu'il contient parce qu'il (l'AMS) implémente un chargeur de classe ("class loader") qui se substitue au chargeur de classes de la machine virtuelle Java.

Il est important de disséquer les éléments qui entrent en compte lors de la provision d'un abonnement et durant l'exécution d'un service. On pourra ainsi cibler et mesurer effectivement les impacts des choix architecturaux. Le temps écoulé entre une souscription et la présence effective des services est composé :

1. du temps de traitement de la suscription à l'UGS ;
2. du temps requis pour créer et assembler (insérer l'intelligence relative aux applications ciblées) dans les AMS (les AMS sont partitionnés par catégorie de services) ;
3. du temps requis par la plate-forme d'agents mobiles pour déplacer l'agent de l'UGS à la machine de l'usager ;
4. du temps requis pour transférer les fichiers des services à partir de l'UCS à la machine de l'usager ou à l'UGS.

Le délai 1 est le délai encouru par le traitement de l'information des usagers sur un serveur public quelconque. Ce délai n'est pas déterminé ni influencé par notre architecture. Il ne nous intéresse donc pas. Le délai 2 est entièrement attribuable à notre application. Le délai 3 est déterminé par la plate-forme pour agent mobile et son implémentation de la migration des agents. Toutefois, notre architecture influe sur ce délai car c'est l'UGS qui crée et décide de la taille des agents. Il est évident qu'il est plus rapide de déplacer un "petit" agent plutôt que d'en déplacer un "gros". Nous devons donc trouver les tailles admissibles de l'AMS pour qu'il puisse être déplacé en deçà d'un certain temps. Le délai 4 est entièrement déterminé par le débit du réseau.

Des évaluations de performance des agents mobiles ont déjà été réalisées, notamment par Ranganathan *et al.* (1998), Gray (1997), Picco (1998), Kotz *et al.* (2000), etc. Ces études essaient en général de déterminer quand il est bénéfique d'utiliser des agents mobiles à la place des autres paradigmes, tels client-serveur, évaluation à distance et code sur demande. Toutefois, ces études sont trop générales et ne servent qu'à présenter une vue d'ensemble des considérations à prendre en compte pour déterminer si l'agent mobile permet un gain de performance. Une analyse adéquate de la performance des agents mobiles doit se faire par application et, pour chaque application, tenir compte de son interaction avec la plate-forme d'agents mobiles.

R. Gray (1997) effectue une analyse de performance détaillée de Agent-TCL, le système pour agents mobiles développé à Dartmouth College. Il prend en compte notamment l'effet de l'algorithme de Nagle sur la latence des connexions TCP. Cette latence influe significativement sur la taille des opérations à effectuer avant que l'approche agent mobile soit préférable à client-serveur. Ranganathan *et al.* (1998) étudient les gains de performance d'un serveur de "chat" sur Internet, qui peut se déplacer. Des outils réseaux sont utilisés pour estimer les délais sur les différents liens. Les données collectées servent à déplacer le serveur et le repositionner de manière à minimiser le délai de transmission des messages. G. Picco (1998) compare les performances de l'évaluation à distance, code sur demande et des agents mobiles dans le cadre de la gestion des réseaux. Il utilise une logique non monotone pour formaliser ces notions.

5.1.2 Conditions environnantes et modèle d'échantillonnage

Le but de nos mesures n'est pas de démontrer un hypothétique gain de performance. Nous voulons déterminer si la performance de l'architecture est acceptable. La latence sur la plupart des implémentations TCP est de 400 millisecondes (200 millisecondes sous Linux). Ceci est dû au fait que les machines attendent que la taille du paquet à transmettre (message véritable ou simplement confirmation de réception) à l'interlocuteur ait dépassé la taille de la fenêtre, ou qu'un paquet allant dans la même

direction transite par la machine. Si aucun de ces événements ne se produit, elle transmet le paquet ou la confirmation de la réception après 200 millisecondes. La latence pour un aller-retour est donc de 400 millisecondes. Toutefois, nous ne souffrons pas de cette latence car la taille des paquets que nous transmettons est toujours supérieure au segment TCP sur Ethernet (1472 octets). De plus, nous utilisons HTTP comme protocole pour la couche application. Nous envoyons donc des données sans attendre de confirmation.

Le Tableau 5.1 résume l'état du réseau au moment des tests. Nous avons utilisé le logiciel PACMon (Site www.abraxis.com) pour mesurer le débit du réseau lors des tests. Le débit du réseau est resté pratiquement inchangé pour tous les tests (variation maximum de 0.5 Mégabits/s). Les machines sont connectées par un réseau Fast Ethernet (100 Mégabits/s).

Tableau 5.1 Conditions du réseau

Latence des connexions TCP	0.4 ms
Débit du réseau	56.35 Mbits/s
Capacité du réseau local (LAN)	100 Mbits/s

Tous les résultats présentés aux Figures 5.1 à 5.3 sont la moyenne de 200 mesures réalisées sur une période de 2 semaines. Les tests ont été effectués de 9 heures du soir à 9 heures du matin durant la semaine et à n'importe quelle heure durant les "week-ends". On bénéficiait ainsi de l'exclusivité des machines et peu d'applications encombrent le réseau aux périodes choisies.

En réalité, nous avons effectué 400 mesures pour chaque point de chaque figure. Toutefois, nous avons ensuite éliminé les résultats situés dans le premier et dernier quartile. Le résultat final est que la variance maximale est de 6.7%. C'est la variance du point 7 (5212-6) de la Figure 5.1. La Figure 5.2 a une variance maximale de 3.7%, la Figure 5.3 une variance maximale de 2.2%.

L'UGS est située sur une machine Solaris 2.6 avec un processeur de 333 MHz. L'UCS est située sur une machine Windows NT 4.0 avec un processeur Intel Pentium II de 400/100 MHz. Quatre autres machines ont été utilisées pour accueillir les AMS. Ce sont toutes des stations Windows NT 4.0 avec des processeurs Intel Pentium II de 266 MHz.

5.1.3 Résultats d'expérimentation

Pour notre implémentation, la taille minimale d'un AMS à vide est de 21097 octets. Il serait ardu (nous n'avons pas réussi à le faire) de réduire la taille d'un AMS programmé en Java en deçà de ce niveau. La grandeur totale d'un AMS est constituée de sa taille à vide, plus la taille des services ou pointeurs aux services qu'il transporte, plus la taille des données des services qu'il transporte également. Donc, avec le chiffre de 21097 comme taille à vide en tête, pour tous nos résultats, nous indiquons uniquement la taille des données à laquelle nous ajoutons celle des pointeurs ou des services. Nous indiquons aussi le nombre de services. Dans chaque cas donc, la taille véritable de l'AMS est 21097 octets, plus la taille des données des services, plus la taille des pointeurs ou services, selon le schéma choisi.

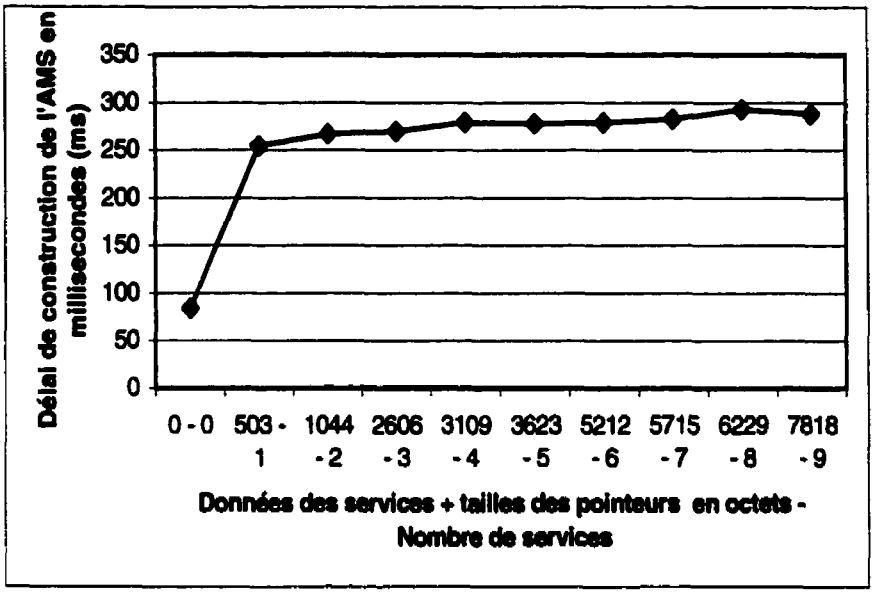
Nous déterminons d'abord le temps nécessaire pour créer un agent. Nous mesurons ensuite le délai de transmission en fonction de la taille de l'agent. Finalement, nous mesurons l'impact de performance dû à l'AMS lors de l'exécution d'un service.

Assemblage de l'AMS

Le temps nécessaire pour assembler les AMS est influencé par :

- la logique qu'il faut configurer pour chaque service ;
- le nombre de services auxquels l'usager a souscrit ;
- le nombre d'AMS à créer.

La Figure 5.1 présente le temps de construction d'un AMS en fonction de la taille des pointeurs et du nombre de services que cet AMS contient.



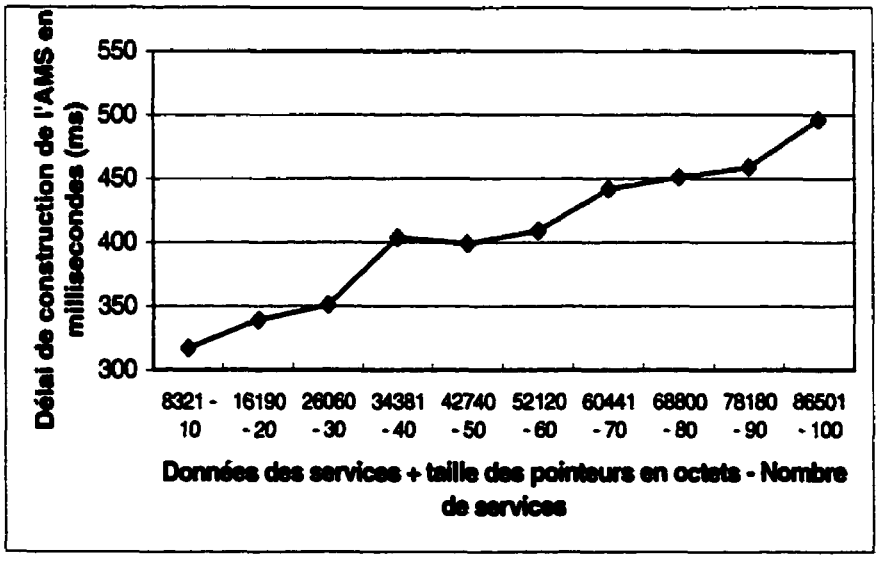


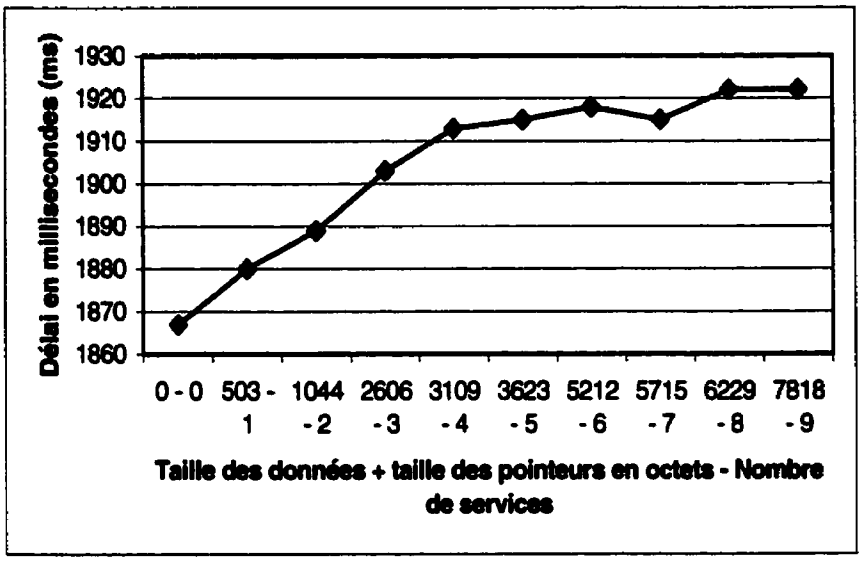
Figure 5.1 Délai de construction pour un AMS contenant un nombre grandissant de services

Comme le montre la Figure 5.1, créer et configurer (insérer de l'intelligence formelle, encoder les données relatives aux services) un AMS prend typiquement entre 250 et 500 millisecondes, même dans le cas où l'AMS devrait contenir des pointeurs à 100 services. Ce délai est faible quand on se rappelle qu'en réalité l'AMS est un proxy dynamique généré à travers l'interface Factory de *Voyager*. *Voyager* utilise le package Reflection de Java pour créer des proxies dynamiques (maléables) de classes. *Voyager* a ainsi un accès direct aux champs internes de l'objet et profite de cet avantage pour accélérer la sérialisation lors de la migration de l'agent.

Délai de transmission de l'AMS en fonction de sa taille

La Figure 5.2 mesure le temps nécessaire pour un agent pour faire un aller-retour sur la plate-forme *Voyager*, en fonction de la taille de cet agent. Quand l'agent arrive sur la machine distante, il exécute une seule instruction, celle pour retourner sur sa plate-forme de départ. Nous avons préféré mesurer l'aller-retour parce que mesurer un délai sur deux ordinateurs différents peut donner des résultats erronés si les deux machines ne sont pas synchronisées.

Le délai pour envoyer et recevoir un agent qui transporte 86500 octets (taille totale = 21097 + 86500) d'information est inférieur à 2.1 secondes. Cette bonne performance est probablement due au fait que *Voyager* implémente la migration faible des agents. Il n'est pas possible d'offrir la migration forte simplement par l'intermédiaire d'une librairie de classes Java, sans modifier Java Runtime. En effet, on ne peut capturer l'état du processus qu'au niveau de l'interpréteur JDK. La migration partielle ne transfère pas l'état du processus de l'agent. Les classes que l'agent implémente sont transférées entre machines et les objets sont recréés sur la machine distante avec les valeurs d'avant le transfert.



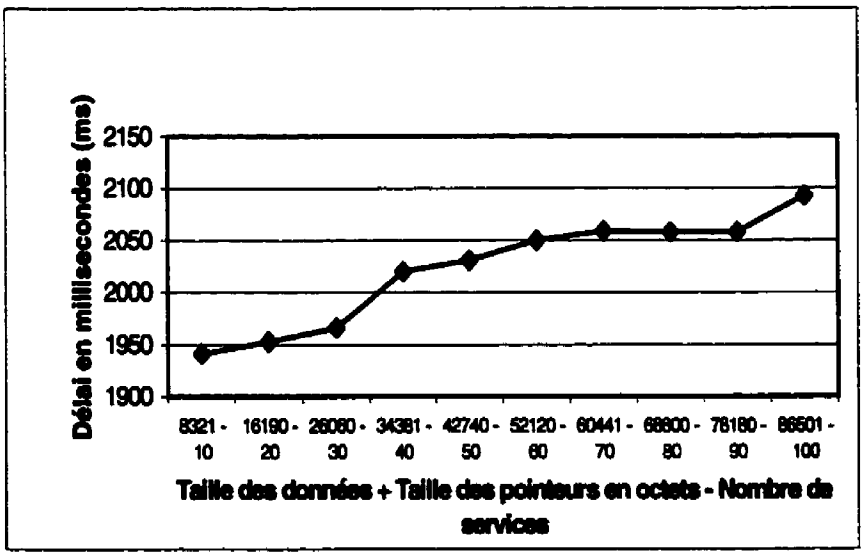


Figure 5.2 Temps nécessaire pour effectuer l'aller-retour dans notre réseau local pour un AMS de plus en plus grand

Les valeurs des champs de tout objet Java sont obtenues par introspection de l'objet en utilisant le package Reflection. Java est l'un des rares langages impératifs qui permette l'introspection, ceci est plus courant pour les langages fonctionnels (LISP, etc.). Bien entendu, avec la migration forte, pas besoin de recréer les objets, le processus en cours d'exécution est lui-même transféré et continue son exécution à ce nœud distant. La migration forte dégrade toujours la performance du système en général (Milojičić, 1999). Les résultats présentés ici suggèrent que la migration faible est une bonne alternative *du point de la vitesse*.

Exécution des services

Lorsqu'un programme est démarré, l'interpréteur Java charge uniquement la classe contenant le point d'entrée du programme (fonction public static main(String args[])). Durant l'exécution, à chaque fois qu'il rencontre une référence à une classe, si cette classe n'a pas encore été résolue, il va la charger en utilisant son chargeur de classes. Le chargeur de classes fouille les répertoires contenus sous la variable environnement ("CLASSPATH"). Toute classe est chargée une et une seule fois dans un programme. Java permet d'écrire des sous-classes de son chargeur de fichier qui seront appelées quand il rencontrera une classe qu'il ne pourra trouver par la variable "CLASSPATH". Ce chargeur de classes personnel peut être écrit pour charger des classes de n'importe quelle source (machine lointaine sur le réseau, serveur HTTP, etc.). Ceci est possible parce que Java est un langage interprété. Ses classes sont compilées séparément, ont une représentation unique en bytecodes et sont portables sur tous les systèmes d'exploitation.

Toutefois, ceci impose un délai puisque l'interpréteur ne peut résoudre directement la classe, et doit recourir à un chargeur de classes développé par le programmeur. La première invocation de classes, qui ne sont pas situées sur le disque dur sous la variable environnement "CLASSPATH", causera un délai variable selon la performance du chargeur de classes implémenté à l'intérieur du programme. Bien entendu, le délai est plus grand si ce chargeur de classes doit transférer les classes à partir

d'une machine distante. Toutefois, nous ne mesurerons pas le délai pour transférer un fichier car celui-ci varie avec le réseau.

La dégradation de performance attribuable au chargeur de fichier de l'AMS est résumée au Tableau 5.2.

Tableau 5.2 Impact sur l'exécution d'un programme par l'AMS

Temps de résolution (chargement) d'une classe contenue dans l'AMS	0.40 ms
Ajout au temps d'exécution d'un service qui utilisent N classes contenues dans l'AMS	N * 0.40 ms
Temps pour transférer un fichier de taille S sur un réseau avec débit D (pas d'attente)	S/D
Ajout au temps d'exécution d'un service quand l'AMS contient juste des pointeurs et doit transférer les fichiers sur le réseau	N * (S/D + 0.40) ms

Un service exécuté à travers l'AMS prendra donc N*0.40, ou N*(S/D +0.40) millisecondes (selon les stratégies employées) de plus pour terminer, comparé au cas où ce service était sur le disque dur local. Quand on transporte les services eux-mêmes, il faut 2500 classes pour qu'il s'ajoute 1 seconde à la durée d'exécution d'un service transporté par l'AMS. La dégradation de performance est pratiquement imperceptible car, elle n'est pas groupée à un endroit particulier du programme et s'étale sur toute la durée d'exécution de l'application. Si les fichiers doivent être chargés à distance, le délai varie avec l'état du réseau.

5.2 Évaluation de la permutation d'agents

Le temps requis pour achever la permutation graduelle est principalement composé du :

- temps nécessaire pour créer le nouvel AMS ;

- temps de transfert des codes exécutables à partir des UCS jusqu'à l'AMS ;
- temps de transfert des données de l'ancien AMS au nouveau ;
- temps de déplacement du nouvel AMS de l'UGS jusqu'au site de l'ancien AMS.

Dans le cas de la permutation abrupte, il faut rajouter en plus des délais de la permutation graduelle, le temps requis pour redémarrer tout service qui aurait été interrompu. Le temps de transfert des données entre les AMS est négligeable, typiquement moins de 20 millisecondes. Ceci est dû au fait que la taille des données à transférer n'est pas importante. La Figure 5.3 présente la courbe de transfert des données sur le réseau dans les mêmes conditions expérimentales que celles de l'analyse de performance.

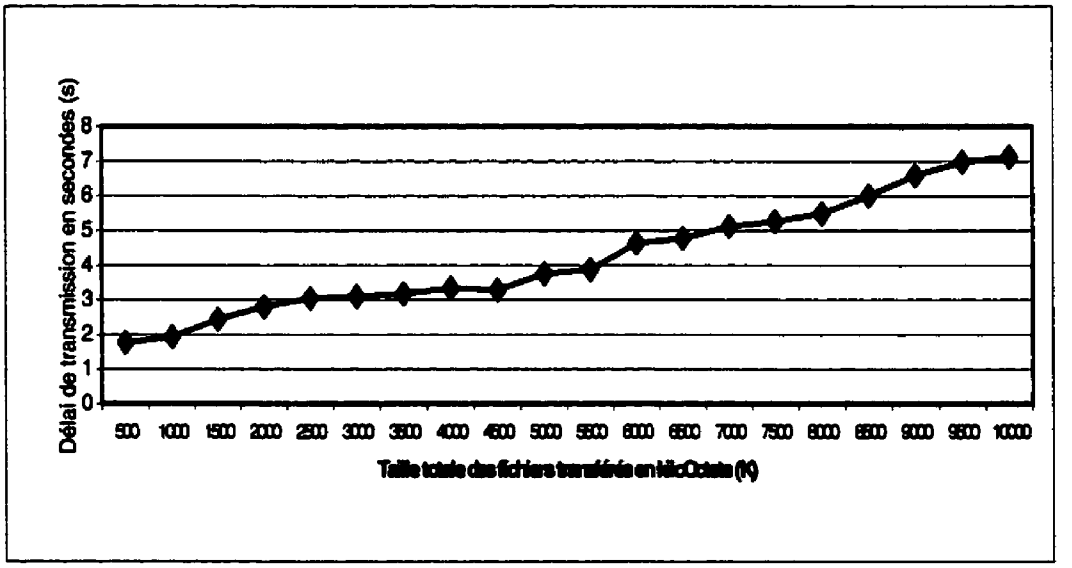


Figure 5.3 Courbe des délais de transmission sur le réseau aux moments des tests

Délai pour la disponibilité

Posons T_C comme le temps de création de l'AMS, T_T le temps de transfert des données, et T_D le Temps de déplacement de l'AMS. Alors, le temps de complétion de la

permutation graduelle pour un AMS contenant 10 services totalisant 1000 Kilo-octets est approximativement de:

$$T_C + T_T + T_D = 300 \text{ ms} + 2032 \text{ ms} + 1150 \text{ ms} < 3.5 \text{ s}.$$

Nous avons volontairement majoré les valeurs et considéré des grandes tailles pour les services. N'empêche, on se retrouve à moins de 3.5 secondes. Mille Kilo-octets est une taille très importante, principalement pour les "palmtops" qui sont limités à quelques Mega-octets. Même dans les autres cas, en allant aux extrêmes des courbes avec 10000 K et 100 services, nous nous retrouvons avec un délai de 500 + 7200 + 1300 < 9.1 secondes. La performance reste acceptable pour ce cas pourtant non réaliste.

Dans le cas de la permutation abrupte, il faut ajouter le temps de démarrage des services qui ont été interrompus au besoin. Toutefois, parce que les services sont démarrés en parallèles dans des "threads", le délai de démarrage d'un nombre N de services est égal au délai de démarrage d'un service. Le temps nécessaire pour arrêter les services est négligeable, i.e. moins de 1 milliseconde par service. Si on considère que n services ont été interrompus et que redémarrer chaque service provoque l'initialisation de 10 classes par service, en se rappelant que le temps de résolution d'une classe par l'AMS est de T_R=0.40 ms, le délai est de :

$$T_C + T_T + T_D + 10*T_R = 300 \text{ ms} + 2032 \text{ ms} + 1150 \text{ ms} + 10*0.40 < 3.5 \text{ s}.$$

Le temps de redémarrage de tous les services inflige une pénalité égale au nombre de classes de ce service qui doivent être chargées au début de son exécution. La durée de l'interruption des services est inférieure à :

$$T_D + 10*T_R = 1150 \text{ ms} + 10*0.40 < 1.5 \text{ s}.$$

Évaluation des requis

Le Tableau 5.3 résume l'évaluation de la permutation graduelle et de la permutation abrupte par rapport aux requis énoncés dans la section 5.1.

Tableau 5.3 Évaluation des stratégies de permutation

	Permutation graduelle	Permutation abrupte
Interruption de service	Aucune	Faible, varie avec le nombre de classes à transférer < 3.5 s. sur Fast Ethernet
Délai pour la disponibilité	Courte durée. Temps de déplacer l'agent et démarrer (résoudre) les classes des services	Faible, même comportement que l'approche graduelle. Ajouter le temps de résoudre les classes
Dépendances systèmes	Aucune	Aucune
Simplicité	Faible	Faible
Performance à grande échelle	Bonne. Les seuls facteurs variables, les temps de déplacement de l'agent et de transfert des services progressent selon des courbes 15/8000 ms/octets et 1/1800 s/Kilo-octets respectivement	Même observation que permutation graduelle. Le temps de démarrage des services est uniforme et négligeable
Impact sur les anciens services	Faible, pas de personnalisations possibles durant la permutation	Aucun

5.3 Mise à jour dynamique de l'AMS

Nous avons vu au chapitre 4 que l'investigation de l'ajout dynamique de services qui ne sont pas en cours d'exécution est d'un intérêt mitigé. Les données d'une analyse de performance dans ces cas là seront certainement similaires à celle obtenues lors de l'évaluation de la permutation d'agent. Nous procédons ici à l'analyse de performance d'un programme qui est mis à jour dynamiquement. Nous comparons ensuite cette performance aux requis énoncés au chapitre 4.

5.3.1 Évaluation de performance

Parce que l'implémentation est basée sur des proxies, chaque appel de méthode sur un objet dynamique résulte en une vérification et une invocation de méthode supplémentaire, si la nouvelle version de la classe implémente une méthode qui a les mêmes noms et arguments que celle appelée. Par contre, si l'appel doit être redirigé vers une méthode différente, chaque appel implique une vérification, un appel et le retrait de la méthode appropriée d'une table. Dans les deux cas, l'appel supplémentaire utilise l'interface *Reflection* qui est plus lente que les invocations directes.

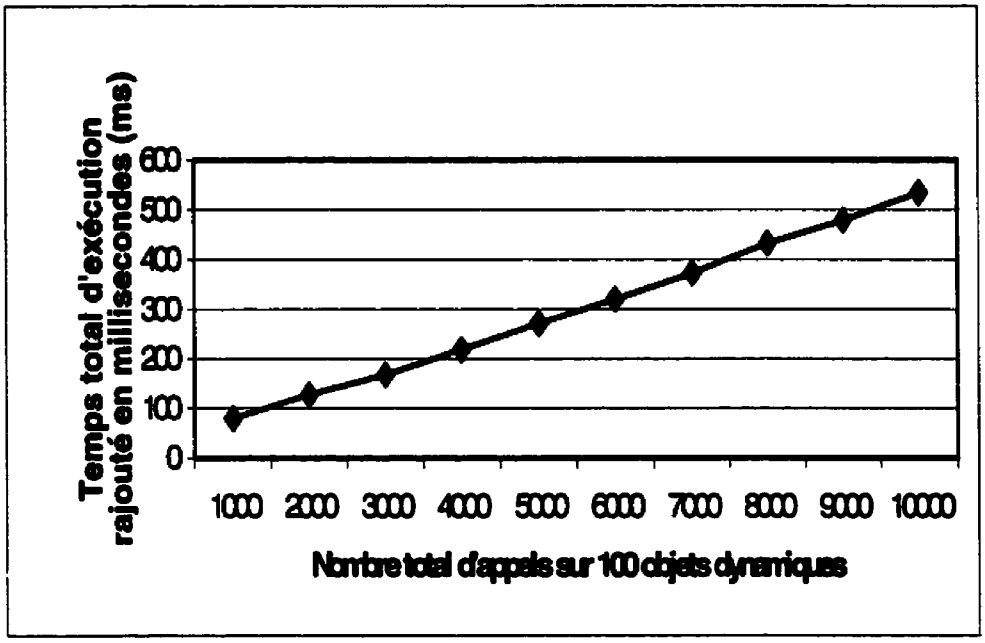
Nous mesurons le temps supplémentaire qu'un programme prend pour achever quand il modifie des objets dynamiques et invoque leurs méthodes. Nous étudions les deux cas : le cas où l' "InvocationHandler" du *proxy* appelle tout simplement la même méthode dans la nouvelle version, et le cas où l'appel doit être redirigé selon une autre politique spécifiée par l'usager. Il est plus approprié de mesurer l'impact de performance de la mise à jour dynamique pour chaque application. Nous expliquons plus loin pourquoi.

Chaque point dans les Figures 5.4 et 5.5 est la moyenne de 100 essais. Dans la Figure 5.4, l'écart-type maximal est de 6.1% et il est de 5.7% dans la Figure 5.5. L'ordinateur des tests était une Sun UltraSparc roulant Solaris 2.6 avec un processeur 333 MHz.

Pour tous les tests, nous avons utilisé 100 classes, chacune implémentant 10 méthodes. Chaque test manipulait 100 objets dynamiques et avait une séquence d'appel différente de celle des autres. La même méthode était invoquée sur un objet dynamique seulement après que toutes les autres méthodes de cet objet aient été appelées. Ces permutations nous ont permis de tempérer les possibles gains de performance dus aux mécanismes de cachage. L'application test était un programme de tri rapide des octets d'un fichier binaire de 73.8 Kilo-octets. L'application prend 5.11 secondes pour achever. Nous avons inséré 100, puis 200, puis 300 et ainsi de suite instructions manipulant les objets dynamiques. Les résultats donnent une idée de la dégradation de performance pour un programme de calcul intense.

La Figure 5.3 montre les effets des redirections des appels quand il s'agit tout simplement d'appeler la même méthode par *Reflection* sur un nouvel objet de la classe. Jusqu'à 1000 appels, la pénalité est inférieure à 0.1 seconde et à 10000 appels est d'un peu plus de 0.5 seconde. Par contre, avec 100000 appels, la pénalité est de 4.7 secondes, doublant effectivement le temps d'exécution du programme. Ces observations renforcent le fait que les mesures doivent être évaluées en fonction de l'application. En effet, 4.7 secondes dans le cas d'un tri rapide double le temps d'exécution. Par contre, pour un programme qui dure 7 minutes ou plus, 4.7 secondes correspondent à une baisse de performance de moins de 0.1% (au lieu de 100% comme dans ce cas ci).

Encore plus important, les applications multimédia (vidéo MPEG, audio MP3, etc.) sur les petites machines d'information portables (Palm Top) exécutent en continu (longue durée de temps) et sont caractérisées par des pointes discrètes de calcul. Ceci est totalement à l'opposé du cas d'un nombre important de calcul continu effectué dans une courte période de temps comme dans le cas des tris ou des multiplications de matrices. Ces brefs calculs impliquent généralement peu d'objets (les applications sont de petites tailles) et le nombre d'appels de méthodes est faible (cent ou moins) par rapport au cas considéré ici.



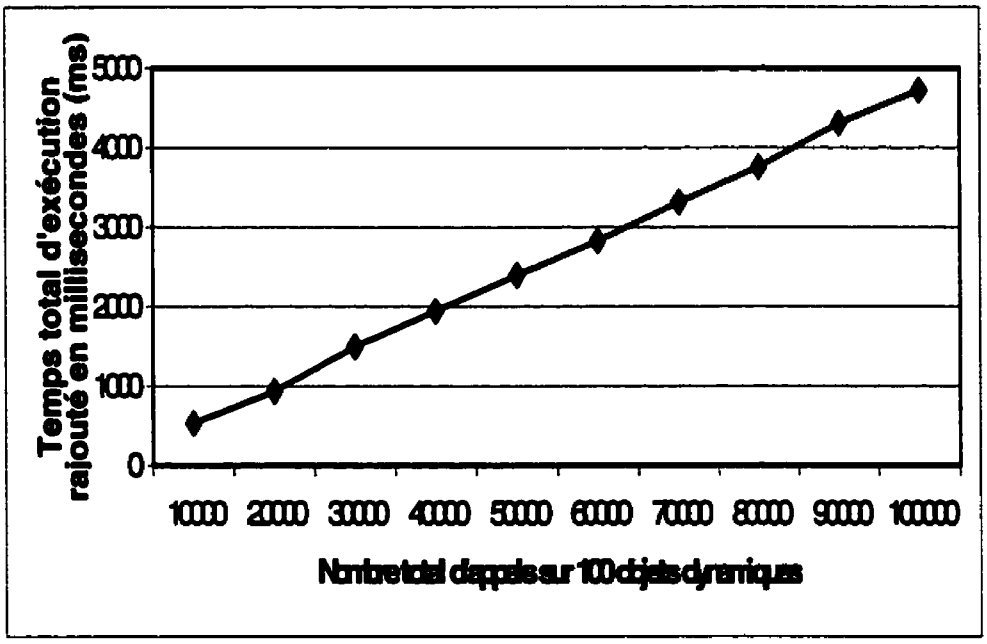


Figure 5.4 Pénalité de performance avec une redirection simple sur les objets dynamiques

La pénalité pour 10 redirections est inférieure à une milliseconde et celle de 100 redirections est de 7.3 millisecondes. La conclusion est évidente ; on ne remarque *pas de pénalité interactive en utilisant les objets dynamiques*. Les exemples de petites applications qui doivent rouler continuellement sur des PalmPilot sont nombreux. Un service de reports de valeurs boursières qui surveille plusieurs marchés dans différents pays (et fuseaux horaires) et qui doit intégrer une nouvelle interface graphique, service de surveillance météo qui rapportent les conditions météorologiques, etc. La mise à jour dynamique évaluée ici est donc tout indiquée pour ces applications là.

La performance dans le cas d'une redirection simple est bonne. La Figure 5.4 présente la pénalité pour une stratégie de redirection plus élaborée, i.e. la méthode où l'appel doit être dévié n'est pas tenu d'avoir le même nom. Toutefois, il doit accepter les mêmes arguments. La pénalité est plus élevée. Avec 10000 invocations, le programme prend près de 2 secondes de plus. Comme pour les autres figures, l'évolution est linéaire. On peut projeter et déduire que, pour 100000 invocations, la pénalité sera de 20 secondes. La baisse de performance est plus notable ici. Toutefois, l'approche offre plus de flexibilité pour la gestion des méthodes. Si la performance est une préoccupation majeure, l'option précédente est préférable; il faut alors réutiliser les mêmes noms de méthodes.

La seconde option devrait être utilisée lorsque le nombre d'invocations prévues est moins élevé. Sa performance est acceptable et elle devrait trouver une niche avec les applications qui roulent pour une longue durée, qui ont des exigences de performance moins élevées et qui tirent profit de sa plus grande flexibilité. Ici en effet, seuls les arguments doivent être identiques d'une fonction à l'autre. L'appel peut donc être dévié vers une méthode quelconque pourvu qu'elle prenne les mêmes arguments.

La performance de la librairie dynamique lorsqu'elle applique une politique de redirection est convenable. Toutefois, elle devrait se dégrader rapidement si l'implémentation permettait des redirections quelconques indépendantes des arguments des méthodes. Il faudrait alors vérifier quels paramètres devrait être éventuellement éliminés, et comment replacer les paramètres dans le bon ordre. Ceci se traduit

concrètement par plusieurs autres vérifications et manipulations, dépendamment des redirections permises.

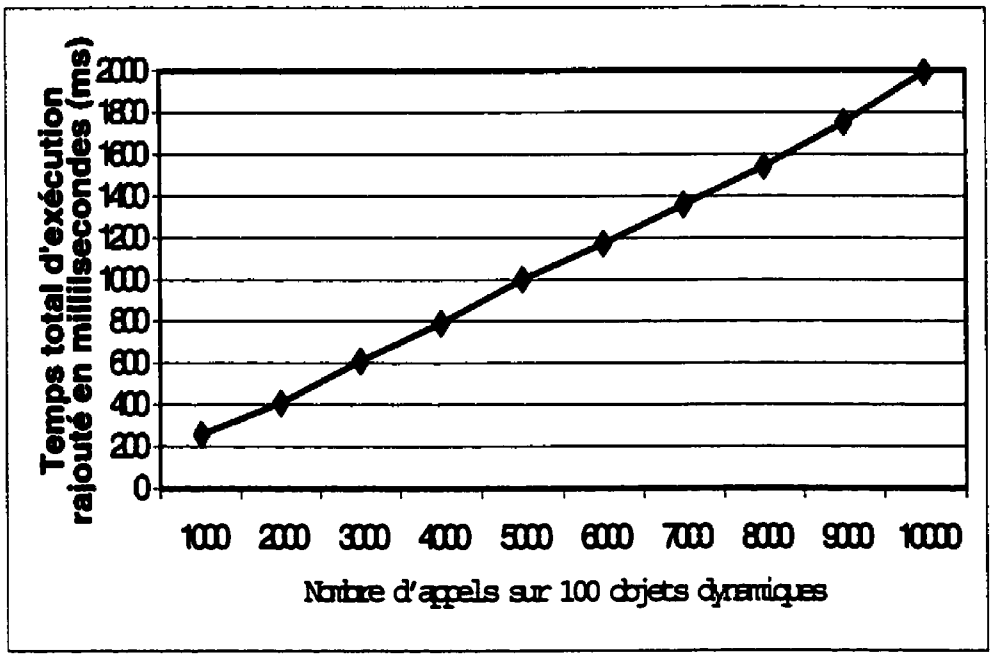


Figure 5.5 Pénalité avec une stratégie élaborée de redirection sur les objets dynamiques

5.3.2 Évaluation des requis

Les mises à jour dynamiques satisfont les requis relatifs à une solution au problème. Il n'y a pas d'interruption de service à cause de la mise à jour. Les données personnalisées par les usagers sont conservées puisque l'AMS n'est pas changé ; ce sont ces services qui sont modifiés. La mise à jour est transparente aux applications. La politique de mise à jour est volontairement minimaliste, et le défaut (redirection à une fonction du même nom) est suffisant dans la plupart des cas. La solution est indépendante de la plate-forme : c'est une librairie, toute machine virtuelle Java standard peut l'utiliser.

Lorsqu'un abonné fait de nouvelles suscriptions, la liste de ces suscriptions est envoyée à l'UGS. Un message de notification contenant la politique de mise à jour des services, la liste des nouveaux services ainsi que la localisation des fichiers et des données est envoyé à l'AMS. Sur réception du message, l'AMS transfère les codes exécutables et les données des nouveaux services. Il inclut les données de ces services dans sa base de connaissance, les indexe et se rappelle qu'il contient des nouveaux services et leurs caractéristiques. Ensuite, il modifie son interface graphique de sorte que l'usager puisse maintenant démarrer, arrêter et personnaliser ces services. Les services existants sont mis à jour selon la politique du developpeur de service qui était contenue dans la notification. Sinon, la mise à jour se fait selon les défauts présentés au chapitre 4.

Plusieurs facteurs influencent le délai entre un abonnement et la disponibilité des services. L'architecture joue un rôle dans le temps nécessaire pour informer l'AMS de la nouvelle souscription et ses caractéristiques (nouvelles versions et nouveaux services). Si S_M est la taille du message à envoyer à l'AMS, S_{TS} la taille totale de tous les nouveaux services plus la taille totale des classes dynamiques des services existants, alors le délai avant qu'un service soit disponible est :

$$T_{Delay}=S_M/T+S_{TS}/T+T_{bind}$$

T est le débit du réseau, et T_{bind} le délai infligé par le chargement dynamique des classes à travers le chargeur de classes de l'AMS. L'AMS démarre autant de "threads" que nécessaires pour transférer les fichiers en parallèle. Les nombres varient avec l'application. Toutefois, par souci de comparaison, S_M était de 633 octets pour un service avec 5 classes (totalisant 23 Kilo-octets) à mettre à jour et une politique de mise à jour pour trois d'entre elles. T_{delay} est de 104 millisecondes dans ces circonstances avec un Fast Ethernet (capacité de 100 Mbits/s) et un débit réel de 52 Mbits/s. L'AMS reçoit les messages et transfèrent les fichiers à travers des sockets Unix. HTTP 1.1 était utilisé pour le transfert des messages tout comme pour le chargement des fichiers.

La solution s'adapte bien à un nombre élevé de services, i.e. le délai pour N services est le délai pour 1 service grâce au parallélisme. Plus encore, dans l'architecture,

les services opèrent et sont modifiés indépendamment les uns des autres. L'évaluation des requis est résumée au Tableau 5.4.

Tableau 5.4 Évaluation de l'implémentation par rapport aux requis

	Objets dynamiques
Impact sur les services existant	Faible, les objets inchangés préservent l'information
Interruption des services	Non
Adaptable à grande échelle	Oui
Simplicité	Les mises à jour sont transparentes à l'application
Délai pour la disponibilité	Bas, les services sont transférés en parallèle
Dépendance vis-à-vis du système	Aucune

	Chargement de nouveaux services et leurs données
Impact sur les services existants	Aucun
Interruption des services	Non, chargement dans des processus poids-légers
Adaptable à grande échelle	Oui
Simplicité	Oui
Délai pour la disponibilité	Faible, parallélisme
Dépendance sur le système	Aucune

Il est aussi possible d'envoyer des agents de mise à jour qui se promèneront d'hôtes en hôtes au lieu d'envoyer des messages. Il faudrait vérifier d'abord si ce schéma permet un gain de performance.

5.4 Synthèse des performances

L'analyse de performance a permis de valider l'implémentation de notre architecture pour la mobilité des services. Les délais encourus avant la disponibilité des services sont faibles. L'impact de performance de l'AMS sur les applications (qu'il contient) est infinitésimal. Il devrait dorénavant être considéré comme négligeable.

La permutation d'agent remplit les exigences énumérées au chapitre 4. Seule la permutation abrupte cause une interruption de service. Les performances des permutations graduelle et abrupte sont comparables, pratiquement égales. Le délai pour la disponibilité des services est faible dans les deux cas. La permutation abrupte expose l'AMS aux pannes du réseau. À titre d'exemple, la permutation d'un agent contenant 10 services totalisant 1000 Kilo-octets prend moins de 3.5 secondes. L'interruption de service lors de transition abrupte est inférieure à 1.5 seconde.

La mise à jour dynamique de l'agent est encore plus rapide à cause des traitements en parallèle et du fait qu'il n'est pas nécessaire de créer un nouvel agent. Le délai pour mettre à jour 5 classes totalisant 23 Kilo-octets est de 104 millisecondes. La dégradation de performance attribuable à la redirection des appels sur les classes dynamiques est faible, notamment dans le cas d'une redirection simple. Dix mille (10000) appels causent une dégradation de moins de 550 millisecondes avec des redirections simples. La perte est de près de 2 secondes avec une politique élaborée dans le cas de 10000 appels.

CHAPITRE VI

CONCLUSION

La téléphonie Internet en combinaison avec la vulgarisation des moniteurs d'information portables offre un nombre spectaculaire d'opportunités. Les coûts sont réduits et des services inconcevables il y a quelque temps peuvent être produits. Les exemples actuels de tels services associent souvent plusieurs technologies (téléphonie, courriel et facturation sur carte de crédit). Des architectures de services permettant la libre concurrence et basées sur une implémentation technique performante et augmentable sont nécessaires afin que la précipitation n'amène à produire des solutions limitées, élaborées pour un rapide et court gain commercial. L'utilisation d'agents mobiles a été proposée dans la littérature. Toutefois, ces propositions étaient limitées aux réseaux téléphoniques et aucune implémentation ne validait les proclamations. Ce mémoire a modifié et étendu les concepts pour proposer des agents mobiles qui regroupent et transportent les services ayant des affinités. De plus, tout service peut être inséré dans l'architecture, qu'il soit de téléphonie ou non.

6.1 Synthèse des travaux et contributions principales

Nous avons spécifié et implémenté une architecture de service qui remplit les exigences de l'introduction et qui est applicable à tout service (de téléphonie ou non) et à toute machine munie d'un interpréteur. La solution a ainsi dépassé le cadre des services avancés de téléphonie. Les éléments principaux de l'architecture sont une unité de gestion des services, une unité de création des services et un agent mobile pour service. Les services sont conservés dans l'UCS. Lors d'un abonnement, l'UGS assemble autant d'agents qu'il y a de classes de services concernées par la souscription. Ces agents se déplacent sur la machine de l'usager (ou son commutateur d'attache dans le cas des services de téléphonie) avec des pointeurs aux codes, ou chargent les codes exécutables

des services avant de se déplacer. Sur la machine de l'usager, ils offrent une interface graphique par laquelle celui-ci peut démarrer, arrêter et personnaliser ses services.

L'agent se déplace sur toute machine de l'usager quand celui-ci le désire. L'accès universel est garanti car l'agent continue à fonctionner même coupé du réseau.

Les autres exigences de l'introduction sont satisfaites. Créer un service revient à le programmer et l'entreposer dans l'UCS. L'AMS se charge de la gestion des services. Tout fournisseur de service accrédité pourrait créer son AMS, ou en partager avec d'autres. L'évolution est indépendante du réseau et les services peuvent être personnalisés. Tout type de service peut être créé : il n'y a aucune restriction à part la nécessité d'utiliser notre librairie si on veut des mises à jour dynamiques. Nous avons résolu ensuite la question de la mise à jour de l'agent et des services. Nous avons d'abord analysé les implications théoriques de deux méthodes de mise à jour, puis explicité leurs implémentations. Les mises à jour se produisent lors d'une nouvelle souscription par un usager qui possède déja un AMS. Les deux méthodes sont la permutation d'agent et la mise à jour dynamique de l'agent. Dans le premier cas, un agent contenant les nouveaux et anciens (nouvelles versions au besoin) remplace l'ancien AMS. Dans le second cas, les nouveaux services et versions sont dynamiquement insérés dans l'AMS. La permutation d'agents a deux variantes : la permutation abrupte et la permutation progressive. Les mises à jour dynamiques sont les plus délicates car l'intégrité du code peut être violée par ces changements.

L'approche de mise à jour dynamique implémentée et décrite au chapitre 4 est une contribution majeure au domaine de l'évolution dynamique des systèmes. En effet, la technique implémentée par notre librairie permet de sélectionner quelles instances mettre à jour et de définir une politique de mise à jour adaptée à chaque application. Une méthodologie de mise à jour peut être définie pour chaque version d'une classe dynamique. En accord avec le paradigme orienté objet, l'unité (granularité) de mise à jour est la classe. Par sa précision (choix des objets à modifier) et sa flexibilité (politique différente pour chaque version d'une classe), notre librairie offre une librairie de mise à jour compacte (9 KiloOctets) et plus efficiente que les approches documentées.

Finalement, nous avons évalué toutes nos implémentations. Il y a une pénalité négligeable (en dessous de la milliseconde par classe) à exécuter les services à travers l'AMS. Le délai pour la disponibilité d'un service est faible et la solution s'adapte bien dans le cas de l'abonnement à un nombre élevé de services. La permutation d'agents a des délais semblables à un premier abonnement. Le délai requis pour arrêter les services est négligeable et l'ajout de temps est dû au temps de déplacement de l'AMS dans le cas de la permutation brusque. Avec la permutation progressive, les deux agents coexistent pour une certaine durée, ce qui n'est pas le cas avec la permutation abrupte. Les délais pour la disponibilité des services sont encore plus faibles dans le cas de la mise à jour dynamique parce qu'il n'est pas nécessaire de créer un nouvel agent ou de charger les anciens services. Dans les deux approches de mise à jour, les données personnalisées sont préservées. Les classes dynamiques imposent une faible pénalité lors de l'exécution du programme. Cette pénalité augmente linéairement avec le nombre d'appels effectués aux classes dynamiques. La pénalité est plus importante quand une politique de redirection autre que celle de défaut est utilisé. La performance du système est très bonne.

6.2 Limitations des travaux et recherches futures

Des implémentations résussies ont guidé, dominé et décidé les divers choix architecturaux et de mise à jour que nous avons adopté. À chacune des étapes, notre méthodologie était donc d'identifier les fonctionnalités requises, puis produire un prototype performant (très rapide) et valide, et enfin d'énumérer les spécifications techniques de l'approche que l'implémentation avait prouvé comme étant la meilleure. Ce faisant, nous avons survolé la théorie et n'avons pas fourni de formalisme pour nos réalisations. Une des voies de recherche future est d'élaborer un formalisme qui prouvera et permettra éventuellement de découvrir des erreurs (généralement indécélables par l'expérimentation) dans les concepts d'AMS et mises à jour dynamiques tels que nous les avons développés. Les méthodes formelles sont embryonnaires dans le domaine de l'informatique mobile. UNITY (McCann et Roman, 1999) est le seul système de raisonnement sur la mobilité du code que nous connaissons. Il est basé sur la logique "π-

calculus" proposé par Milner *et al.* (1992). Le domaine de la formalisation de la validité des mises à jour dynamiques est plus dévelopé et utilise la logique des prédicats courante.

Notre implémentation utilise des mesures de sécurité (codage notamment) uniquement lorsque les oublier invaliderait la solution. La prochaine étape serait d'intégrer les mécanismes de sécurité existants dans l'industrie dans l'architecture. Nous pensons notamment à des mesures spécifiques pour protéger le code objet des services transporté par l'AMS. Ceci est essentiel pour un environnement industriel, sinon l'hôte pourrait capturer l'agent et récuperer les services, ou recupérer les données enregistrées sur disque dur. À ce moment là, il peut alors se dispenser de l'AMS. Bien entendu, toutes les questions de sécurité relatives aux agents sont aussi applicables à l'AMS.

Pour les petites machines tels que les "palmtops" qui ne disposent que de quelques MégaOctets de mémoire, la taille de l'AMS et ses services pourrait devenir trop importante dans certaines criconstances. Ces machines réservent quelques kiloOctets à l'AMS, car elles doivent aussi accommoder les autres applications. Il faut alors envisager des mécanismes de partitionnement de l'intelligence et des services transportés par l'AMS. Plus spécifiquement, les techniques du modèle client-serveur étendu, où les fonctionnalités du serveur sont déplacés au client et vice-versa selon le cas, pourraient être appliqués ici. Il faut donc trouver des moyens de partionner l'intelligence et les fonctionnalités de l'AMS. Face à des limitations de mémoire, puissance du processeur ou même connectivité, l'AMS pourrait s'importer partiellement sur la machine cible et garder ses autres fonctionnalités à une machine (celle du SMU par exemple) et les utiliser ensuite en client-serveur. Les interpréteurs et systèmes d'exploitation permettent d'évaluer les limitations mémoires. Seuls les éléments indispensables sont déplacés sur la machine cible. Le reste suivra si les conditions s'améliorent. L'AMS devrait être assez intelligent pour gérer lui-même son partitionnement.

L'implémentation et la validation de l'architecture présentées dans ce mémoire constituent les fondements indispensables pour la réalisation effective du paradigme que nous avons modifié : celui de l'utilisation d'agents pour l'approvisionnement en services. À partir de cette base éprouvée, les travaux de recherche se multiplieront certainement.

BIBLIOGRAPHIE

Abu-Hakima S., Liscano R. et Impey R., "A Common Multi-Agent Testbed for Diverse Seamless Personal Information Networking Applications", IEEE Communications, Mars 1998, pp. 68-74.

Armstrong J., R. Virding, C. Wikström et M. Williams, "Concurrent Programming in Erlang", Second Edition, Prentice Hall, 1996.

Baumannn J., Kohl F., Rothermel K. et Strasser M., "Mole – Concepts of a Mobile Agent System", Mobility Processes, Computer and Agents, ACM Press, Addison Wesley, 1998, pp. 536-556.

CCITT Recommandation Z.100, "Specification and Description Language (SDL)", Novembre 1999, Genève, Suisse.

Claypool M., Coates T., Hooley S., Shea E. et Spellacy C., "Video Performance in Java", In Proceedings of the Information Resources Management Association Conference, Anchorage Mai 2000, pp. 1-6.

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

Drossopoulou S., Valkevych T., Eisenbach S., "Java Type-Soundness Revisited", Technical Report Imperial College of Science, Technology and Medicine, Avril 2000.

Evans H. et Dickman P., "Zones, Contracts And Absorbing Change: An Approach To Software Evolution", In Proceedings of Object Oriented Programming Systems and Languages Conference, 1999, pp. 415-434.

Finin T., Labrou Y. et Peng Y., "Mobile Agents Can Benefit From Standard Efforts on Interagent Communication", IEEE Communications, Mars 1998, pp. 50-55.

Franz M., "Dynamic Linking of Software Components", IEEE Computer, Mars 1997, pp. 74-81.

Freund S. et Mitchell J. C., "A Type System For Object Initialization In The Java Bytecode Language", In Proceedings of Object Oriented Programming Systems and Languages Conference, 1998, pp. 310-327.

Gamma E., Helm R., Johnson R. et Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

Glitho R., "Emerging Alternatives to Today's Advanced Service Architectures for Internet Telephony: IN and Beyond", Computer Networks Journal, À paraître en Février 2001.

Glitho R., "Advanced Services for Internet Telephony: a Critical Overview", IEEE Network, Juillet-Août 2000.

Glitho R., Pagurek B., Tang J., White T., "Management of Advanced Services in H.323 Internet Protocol Telephony", IEEE Infocom Conference, March 2000.

Glitho R., Wang A., "A Mobile Agent-Based Service Architecture for Internet Telephony", International Switching Symposium, Birmingham May 2000

Glushko R., Tenebaum J. et Meltzer B., "An XML Framework for Agent-Based E-Commerce", Communications of the ACM, Vol 42, No 3, Mars 1999, pp. 106-114.

Graham P., "Ansi Common LISP", Prentice Hall, 1995.

Gray R., "Agent Tcl: A Flexible And Secure Mobile-Agent System", Ph. D. Thesis, Dartmouth College, 1997.

Greenberg M., Byington J., Holding T. et Harper D., "Mobile Agents and Security", IEEE Communications, Mars 1998, pp. 74-85

Griffeth N. D. et Lin Y-J., "Extending Telecommunications Systems: the Feature Interaction Problem", Computer, Vol. 26, No. 8, pp 14-18, August 1993.

Gruber T. R., "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University, Voir Site Ontologies.

Handley D., "SIP: Session Initiation Protocol", RFC 2543, Internet Engineering Task Force, Mars 1999.

Hjálmtýsson G. et Gray R., "Dynamic C++ Classes, A Lightweight Mechanism to Update Code in a Running Program". In Proceedings of the Usenix Annual Technical Conference, New Orleans, Louisiana, Juin 1997.

ITU-T, "Recommendation H.323, Packet-Based Multimédia Communications Systems", Genève, Septembre 1999.

ITU-T, "Recommandations X.680-X.683, Specification Of Abstract Syntax Notation Number One (ASN.1) ", 1997.

Jackson M. et Zave P., "Distributed Feature Composition: A Virtual Architecture For Telecommunications Services", IEEE Transactions on Software Engineering, Vol 24, No 10, Octobre 1998.

Jing J., Helal A. et Elmagarmid A., "Client-Server Computing in Mobile Environments", ACM Computing Surveys, Vol. 31, No 2, 1999, pp. 117-157.

Johansen D., Fred Schneider And Robbert V. Renesse, Tacoma Project Pages, 2000, http://www.cs.uit.no/dos/tacoma/index.html.

Joseph A., Tauber D. et Kaashoek M., "Mobile Computing with the Rover Toolkit", IEEE Transactions on Computer Systems, 1997.

Joshi A. et Singh M. P., "Multiagent Sytems on the Net", Communications Of The ACM March 1999/Vol 42, No 3, pp. 38-40.

Karmouch A. et Pham V. A., "Mobile Software Agents: An Overview", IEEE Communications, July 1998, pp. 26-37.

Khrishnamoorthy C. S. et Rajeev S., "Artificial Intelligence and Expert Systems for Engineers", CRC Press, 1996.

Kotz D., Jiang G., Gray R., Cybenko G., Peterson R., "Performance Analysis of Mobile Agents for Filtering Data Streams on Wireless Networks", Technical Report TR2000-366, Dartmouth College, Mai 2000.

Labrou Y. et Finin T., "A Proposal for a new KQML Specification", Technical Report CS 97-03, University of Maryland Baltimore County.

Maes P., Guttman R. H. et Moukas A. G., "Agents That Buy And Sell", Communications of the ACM, March 1999/Vol 42, No 3, pp. 81-91.

Malabarba S., Pandey R., Gragg J., Barr E. et Barnes J.F., "Runtime Support for Type-Safe Dynamic Java Classes", In Proceedings of the ACM OOPSLA Conference, Antibes, France, 2000.

McCann P. J., Roman G.-C., Modeling Mobile IP in Mobile UNITY, À paraître dans ACM Transactions on Software Engineering and Methodology, disponible comme rapport technique à http://swarm.cs.wustl.edu/cgi-bin/pubs/papers

Milner R., Parrow J. et Walker D., A Calculus for Mobile Processes I., Information and Computation 100, Vol. 1 Sept. 1992, pp. 1-40.

Milojičić D., Mobility Processes, Computer and Agents, ACM Press, Addison Wesley, 1998, pp. 451-456.

Oreizy P., Medvidovic N. et Taylor R., "Architecture-Based Runtime Software Evolution", In Proceedings of the International Conference on Software Engineering, 1998, pp.177-186.

Outtagarts A., Kadoch M. et Soulhi S., "Client-Server and Mobile Agent: Performance Comparative Study In The Management of MIBS", First International Workshop on Mobile Agents for Telecommunication Applications, World Scientific, 1999.

Pelletier S.-J., Pierre S. et Hoang H. H., "ISAME : Une Architecture Multi-Agent de Recherche d'Information", INFOR, Vol. 38, No 2, Mai 2000, pp. 65-91.

Picco G. P., "Understanding, Evaluating, Formalizing and Exploiting Code Mobility", Ph.D. Thesis, Politecnico Di Torino, 1998.

Powell M. et Miller B., "Process Migration In Demos/MP", Mobility, Processes, Computer and Agents, ACM Press, Addison Wesley, 1998, pp. 29 – 38.

Ranganathan M., Acharya A., Sharma S. et Saltz J., "Network-Aware Mobile Programs", Mobility Processes, Computer and Agents, ACM Press, Addison Wesley, 1998, pp. 568-581.

Segal M., Frieder O., "On-The-Fly Program Modification: "Systems for Dynamic Updating", IEEE Software, Mars 1993, pp. 53-65.

Shoham Y., "Agent-Oriented Programming", Journal of Artificial Intelligence, 60(1), pp. 51-92, 1993

Wetherall D. J., "Service Introduction in an Active Network", Ph.D. Thesis, Massachussets Institute of Technology, 1999.

White J., Mobility Processes, Computer And Agents, ACM Press, Addison Wesley, 1998, pp. 461-492.

Wikström Å., "Functional Programming Using Standard ML", Prentice Hall, 1987.

Web Sites

CITRIX	http://www.citrix.com
MOA	http://www.camb.opengroup.org/RI/java/moa/index.html
MSNBC	http://www.msnbc.com/news/457651.asp
Ad Astra	http://www.JumpingBeans.com
Aglets	http://www.trl.ibm.com/aglets
Alexa	http://www.alexa.com
Disa	http://www.disa.org
Enchères	http://auction.eecs.umich.edu
	http://www.ebay.com/aw
	http://www.onsale.com
	http://www.auctionet.com
Excite	http://live.excite.com/
E-Watch	http://www.ewatch.com/
Frictionless	http://www.frictionless.com
Jango	http://www.jango.com
Jini	http://www.sun.com/jini
Mole	http://mole.informatik.uni-stuttgart.de
MySimon	http://www.mysimon.com
Obliq	http://research.compaq.com/SRC/publications/cartoons/src-rr-122.html
Personalogic	http://www.personalogic.com
"Push"	http://jm.acs.virginia.edu/department/org/atg/techtalk/"Push"
	http://www."Pointcast".com

Sumatra	http://www.cs.umd.edu/~acha
Tacoma	http://www.cs.uit.no/forskning/DOS/Tacoma
Voyager	http://www.objectspace.com
WBI	http://www.almaden.ibm.com/cs/wbi/
ZdNet	http://www.zdnet.com/zdi/pview/pview.cgi

KIF Spécification	http://logic.stanford.edu/kif/specification.html
JKQML	http://www.alphaworks.ibm.com/aw.nsf/techmain/jkqml
FIPA-ACL	http://www.nortelnetworks.com/products/announcements/fipa
JKP	http://www.csee.umbc.edu/kif/jkp
Xerces, XML-Java	http://xml.apache.org
Ontologies	http://agents.umbc.edu/aw/Topics/Communicative_Agents/ Ontologies/index.shtml
Cryptographie	http://java.sun.com/jce/index.html
Securité	http://java.sun.com/docs/books/tutorial/security1.2/ summary/apicore.html
PACMon	www.abraxis.com
Voyager	www.objectspace.com
Grasshopper	www.grasshopper.de