Titre: Title:	Sécurité des agents mobiles : protocole d'enregistrement d'itinéraire par agents coopérants
Auteur: Author:	Guillaume Allée
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Allée, G. (2001). Sécurité des agents mobiles : protocole d'enregistrement d'itinéraire par agents coopérants [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6952/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6952/
Directeurs de recherche: Advisors:	Samuel Pierre, & Roch Glitho
Programme: Program:	Génie électrique

UNIVERSITÉ DE MONTRÉAL

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE D'ENREGISTREMENT D'ITINÉRAIRE PAR AGENTS COOPÉRANTS

GUILLAUME ALLÉE
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)
FÉVRIER 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE D'ENREGISTREMENT D'ITINÉRAIRE PAR AGENTS COOPÉRANTS

présenté par: ALLÉE Guillaume

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

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

M. DAGENAIS Michel, Ph. D, Président

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

M. ROCH Glitho, M.Sc., membre et co directeur

M. BERNARD Jean-Charles, Ph. D., membre

REMERCIEMENTS

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

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

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

RÉSUMÉ

L'apparition des réseaux étendus a permis de concevoir des applications réparties intéressantes ayant accès en tout temps à l'ensemble des ressources que propose le réseau. La programmation par agent mobile semble une alternative intéressante à celle de client-serveur. L'idée est de remplacer l'envoi des données vers le programme par la mobilité d'un agent se déplaçant sur l'hôte qui possède des données pertinentes. Cependant, certaines applications comme le commerce électronique ne pourront être effectuées par des agents mobiles que si l'architecture est sécuritaire. Les menaces sont d'autant plus importantes pour l'agent que la plate-forme contrôle son environnement d'exécution. Ainsi, une plate-forme peut empêcher un agent mobile de se déplacer vers sa prochaine destination. Le problème consiste donc à vérifier et à enregistrer les identités des plates-formes qui ont exécuté un agent mobile, ceci de manière sûre. Ce problème est difficile à résoudre avec un agent seul car les données que cet agent transporte sont susceptibles d'être manipulées par des plates-formes malveillantes. Une solution proposée pour détecter cette attaque est un protocole d'enregistrement d'itinéraire par agents coopérants.

Ce mémoire a pour but principal de concevoir et d'implémenter un protocole d'enregistrement d'itinéraire et de le valider. Pour y parvenir, nous avons donc élaboré un protocole d'enregistrement d'itinéraire en utilisant le concept d'agents coopérants. Nous avons décrit de manière précise le comportement de chacun des deux agents coopérants ainsi que la procédure de décision pour établir les identités des coupables lorsqu'une attaque est détectée, ce qui n'avait pas été fait avec le protocole d'enregistrement d'itinéraire actuellement disponible dans la littérature.

Pour démontrer les propriétés de sécurité de notre protocole, nous avons passé en revue l'ensemble des attaques possibles. Pour chacune, nous avons décrit le moment de la détection de l'attaque et, lorsque cela était possible, la désignation des coupables. Certaines attaques que notre protocole détecte étaient supposées impossibles par le

précédent protocole d'enregistrement d'itinéraire et elles n'étaient pas détectées. La différence en terme de capacité de détection des deux protocoles correspond donc à ces attaques. En outre, nous avons quantifié la probabilité qu'il y ait une attaque non détectée par le précédent protocole mais détecter par notre protocole. Enfin, nous avons implémenté notre protocole et le précédent sur une plate-forme d'agents mobiles. Nous avons choisi, pour tester ces implémentations, une application d'agent mobile de magasinage ayant un itinéraire non-prédéfini. C'est une application classique utilisant le paradigme agent mobile où l'enregistrement d'itinéraire est critique.

Nous avons testé notre protocole sur l'ensemble des attaques possibles et les résultats ont été conformes à nos prévisions en terme de désignation des coupables et en termes de moment de détection de l'attaque. Nous avons mesuré et comparé trois implémentations (l'agent de magasinage avec notre protocole, l'agent de magasinage avec l'ancien protocole et l'agent de magasinage seul). Les mesures que nous avons prises concernaient le temps d'exécution de l'agent de magasinage sur chaque plate-forme, le temps d'exécution total et la charge réseau utilisée. Ces mesures sont caractéristiques de certains des avantages des agents mobiles. Les résultats montrent que notre protocole a un coût qui est très légèrement supérieur par rapport au précédent protocole, aussi bien au niveau du temps d'exécution que de la charge réseau utilisée. Comme nous avons démontré que notre protocole détecte plus d'attaques, il en résulte qu'il est le plus efficace des deux. La différence de coût entre un agent de magasinage utilisant notre protocole et un agent de magasinage seul est assez grande aussi bien au niveau du temps d'exécution qu'au niveau de la charge réseau. On peut donc en déduire que l'utilisation de notre protocole ne se recommande pas dans des applications où ces coûts ont une importance critique.

ABSTRACT

The emergence of wide area network (WAN) makes possible to program distributed applications that can access network resources at any time. Mobile agent programming is an alternative to client-server. The idea is to replace the interactions between both entities by the mobility of an agent that migrates to hosts providing interesting data. However, some applications like electronic commerce would be implemented with mobile agents only if the architecture is secure. The threats are all the more important since the platform controls the agent's execution environment. Thus, a platform can prevent a mobile agent from moving to the next host. The problem consists in verifying and recording the identities of the platforms, which have executed the agent. This problem is hard to solve with only one agent because malicious hosts can alter the agent's data. A solution to detect these attacks is to use co-operating agents in a protocol to record and verify the itinerary.

The purpose of this thesis is to design and implement a protocol to record the loose itinerary of a mobile agent. For that purpose, we have elaborated a protocol to record itinerary with co-operating agents. We have precisely described the behavior of the two co-operating agents and the procedure to find the culprit when an attack is detected; this is not done in the protocol currently available in the literature.

In order to demonstrate the security properties of our protocol, we have listed the potential attacks. For each of them, we have described the moment of the detection and, when possible, the designation of the culprits. Some attacks that our protocol detects were supposed impossible by the previous protocol and they were not detected. The difference of detection power between the two protocols corresponds to these attacks. Furthermore, we have quantified the probability that an attack, which is not detected by the previous protocol but detected by our protocol, occurs. Finally, we have implemented our protocol and the previous one on a mobile agent system. To test our

implementations, we chose a shopping agent application with a loose itinerary. This is a classical mobile agent application where itinerary recording is important.

We tested our implementation with all possible attacks and the results were compliant with our theory in terms of culprit designation and moment of detection. We have measured and compared three implementations (shopping agent with our protocol, shopping agent with the previous protocol, shopping agent alone). We have measured the execution time on each platform, the total execution time and the network load. Theses are characteristics of some advantages of mobile agent technology. Results show that the cost of our protocol is hardly higher than the previous one for the execution time and the network load. As we demonstrated that our protocol detects more attacks, then it is more efficient than the previous one. The difference between the cost of our protocol and the shopping agent alone is quite important for the execution time and the network load. Then the use of our protocol is not recommended for applications were theses costs are of great importance.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE I
INTRODUCTION

L'apparition des réseaux étendus a permis de concevoir des applications réparties intéressantes ayant accès en tout temps à l'ensemble des ressources et de l'information que propose le réseau. Cependant, la programmation de telles applications demeure difficile. Le problème principal réside dans le caractère statique de l'architecture client-serveur qui est souvent mal adapté à l'aspect dynamique et à la diversité des systèmes ouverts à l'échelle d'Internet. La programmation par *agent mobile* (AM) semble une alternative intéressante à l'approche de client-serveur. L'idée est de remplacer l'envoi des données vers le programme par la mobilité d'un agent qui va se déplacer sur l'hôte qui possède des données pertinentes. Cependant, certaines applications comme le commerce électronique ne pourront être effectuées par des agents mobiles que si l'architecture est sécuritaire, préoccupation du présent mémoire. Dans ce chapitre, nous allons d'abord introduire les concepts de base qui permettent de définir notre problématique de recherche; par la suite, nous exposerons les objectifs visés, pour enfin esquisser le plan du mémoire.

1.1 Définitions et concepts de base

Dans « agent mobile », il y a deux concepts : celui d'agent et celui de mobilité. Greenberg et al. (1998) définissent un *agent* comme un logiciel qui est autonome, qui a un ou plusieurs buts et un ensemble de compétences et qui peut, au besoin collaborer et communiquer avec d'autres agents ou utilisateurs.

Un agent est *mobile* lorsqu'il peut être transporté d'une machine à l'autre sur un réseau. Un agent mobile ne se transporte pas tout seul : c'est la plate-forme où il s'exécute qui va l'envoyer vers une autre plate-forme à travers le réseau. L'ensemble des plates-formes où l'agent est exécuté est appelé *itinéraire de l'agent*.

Une *plate-forme* est un programme à la fois client et serveur qui permet à l'agent de s'exécuter. C'est la plate-forme qui reçoit l'agent et qui l'envoie. La plate-forme doit être installée sur tous les hôtes sur lesquels l'agent s'arrête. Elle doit aussi implémenter entre autres des services de communication par le réseau.

Parmi les avantages des agents mobiles par rapport à l'architecture client-serveur, on peut citer l'exécution asynchrone, l'autonomie, la réduction de la charge du réseau, la facilité de mise à jour d'agent. L'ensemble de ces avantages fait qu'il est intéressant de développer des logiciels avec les agents mobiles.

Le concept d'agent mobile intègre deux participants : le propriétaire de l'agent et celui de la plate-forme. Chacun veut avoir des garanties que l'autre ne pourra pas l'attaquer. D'un côté, on a la plate-forme qui ne connaît pas toujours le programme d'agent mobile qu'elle reçoit et qu'elle doit exécuter. Il faut donc des techniques de *protection de la plate-forme.* Il est nécessaire de protéger l'environnement d'exécution et les ressources de la plate-forme contre plusieurs types d'*attaques* provenant d'un agent mobile. D'un autre côté, le propriétaire d'un agent ne veut pas qu'une plate-forme puisse modifier l'agent ou les résultats qu'il a obtenus sur d'autres plates-formes. Pour parer à ces attaques, il faut trouver des techniques de *protection de l'agent.*

1.2 Éléments de la problématique

Comme nous l'avons dit précédemment, deux types de problèmes se posent en ce qui concerne la sécurité dans le concept d'agent mobile. D'un coté, on veut protéger la plate-forme contre des agents malveillants et d'un autre, on veut protéger l'agent contre la plate-forme qui l'exécute.

Les attaques d'un agent contre une plate-forme sont variées. Un agent hostile peut tenter d'avoir accès à des ressources sans autorisation de la plate-forme,

consommer trop de ressources (comme l'UCT) ou tenter de se faire passer pour un autre agent. Pour protéger la plate-forme, certains chercheurs ont développé plusieurs techniques dont la majorité est basée sur les approches de protection des systèmes conventionnels. D'autres chercheurs ont proposé des concepts nouveaux tel le code avec preuve qui consiste à prouver certaines propriétés sur le code d'un agent comme la protection de la mémoire et à fournir une preuve de ces propriétés. Quand l'agent arrive sur une nouvelle plate-forme, celle-ci vérifie que la preuve est adéquate par rapport au code de l'agent.

Les attaques d'une plate-forme sont encore plus importantes car la plate-forme a accès à toutes les composantes de l'agent et peut les modifier s'il n'y a pas de techniques pour les détecter ou, mieux, pour les empêcher. On peut citer comme attaque l'altération qui modifie le comportement d'un agent ou qui modifie les résultats de l'agent obtenus sur une autre plate-forme, le déni de service où la plate-forme refuse d'effectuer un service demandé par l'agent (par exemple de l'envoyer vers une autre plate-forme).

Les techniques de protection de l'agent proposent souvent de détecter une attaque plutôt que de les rendre impossible. À ce titre, on peut citer une technique de protection qui permet d'enregistrer l'itinéraire d'un agent par un autre agent et de détecter si une plate-forme n'a pas envoyé l'agent là où celui-ci le demandait. En effet, il est important de savoir si l'agent est bien allé sur toutes les plates-formes qu'il voulait visiter, par exemple dans le cas de recherche d'un meilleur prix sur un produit donné. Cette technique a quand même quelques faiblesses au niveau de la détection de certaines attaques. De plus, elle n'a pas été implémentée ni mesurée. Or, il est important de mesurer le coût des techniques de protection car elles ne doivent pas contrebalancer l'avantage que procurent les agents mobiles vis à vis de l'architecture client-serveur.

Certaines techniques de protection de l'agent comme le calcul de fonctions cryptographiques permettent d'empêcher une attaque éventuelle de la part de la plate-forme contre l'agent. En effet, cette dernière calcule une fonction dont elle ne peut comprendre le but et donne un résultat crypté que seul le propriétaire de l'agent peut décoder. Une attaque de la plate-forme demeure possible : comme la plate-forme

contrôle l'ensemble des constituants de l'environnement d'exécution, elle peut modifier, au hasard, certaines entrées, modifier l'ordre des instructions ou même modifier les instructions en tant que telles. Cependant, le fait qu'elle calcule une fonction cryptographique et donc qu'elle ne comprenne pas la fonction de l'agent l'empêche d'effectuer une attaque dans un but précis. Cette attaque sera donc beaucoup moins dangereuse. De plus, elle sera peut-être détectée par le propriétaire de l'agent lorsqu'il va décrypter le résultat envoyé par l'agent. Cependant, on ne connaît pas encore suffisamment d'algorithmes pour pouvoir coder tous les types de fonctionnalité d'un agent mobile de cette manière.

1.3 Objectifs de recherche

L'objectif principal de ce mémoire est de concevoir un mécanisme de sécurisation à la fois robuste et efficace permettant de protéger les agents mobiles contre d'éventuelles attaques sur leur itinéraire qui proviendraient de plates-formes malveillantes. De manière plus spécifique, ce mémoire vise à :

- Concevoir un protocole de protection d'agents basé sur l'enregistrement d'itinéraire et la coopération entre agents, en s'inspirant d'un protocole existant que nous chercherons à améliorer.
- Évaluer la sécurité de ce protocole en déterminant le type d'attaques qu'il détecte, ceci dans le but de comparer cette capacité de détection avec celle du protocole existant dans la littérature.
- Implémenter sur une plate-forme d'agents mobiles notre protocole, ainsi que l'ancien protocole avec une application où l'enregistrement d'itinéraire est critique.
- Effectuer des tests pour voir si l'implémentation détecte bien les attaques qu'elle devrait détecter et mesurer le coût de notre protocole en le comparant avec celui de l'ancien protocole.

1.4 Plan du mémoire

Le chapitre 2 expose la problématique de la sécurité dans le paradigme agent mobile et les principales méthodes de résolution proposées par les chercheurs. Le chapitre 3 décrit le protocole d'enregistrement d'itinéraire original, puis notre amélioration, ainsi qu'une description des attaques et le comportement des deux protocoles dans chacun des cas. On y fait aussi une évaluation quantitative des avantages de notre protocole par rapport à l'ancien protocole d'enregistrement d'itinéraire. Le chapitre 4 présente les détails de notre implémentation ainsi que les mesures que nous avons effectuées sur cette implémentation. Nous analysons également ces résultats. En guise de conclusion, le chapitre 5 résume les principaux résultats que nous avons obtenus, fait état des limitations de l'implémentation du protocole et donne des indications en vue de recherches futures.

CHAPITRE II
SÉCURITÉ DES AGENTS MOBILES

L'informatique n'a cessé d'évoluer depuis sa naissance et les systèmes se sont peu à peu organisés en réseau. L'apparition de la mobilité du code en informatique est un nouveau pas dans cette évolution et elle a donné lieu à de nombreuses recherches aussi bien dans les milieux universitaires que corporatistes. En effet, plusieurs applications basées sur les agents mobiles ont vu le jour, entre autres, en administration de systèmes, en télécommunications, en commerce électronique où l'agent cherche le meilleur prix pour un produit (Goutet, 2000; Lenou, 2000). Cependant, pour que ces applications atteignent leur plein potentiel, il faudrait qu'elles puissent se déployer sur des plates-formes d'agents mobiles sécuritaires. Dans ce chapitre, nous analysons la question de la sécurité des agents mobiles. Après avoir défini les concepts de base du domaine, nous examinerons les techniques de protection de la plate-forme puis celles de protection de l'agent. Nous terminerons par une synthèse des problèmes ouverts.

2.1 Caractérisation de la mobilité et des menaces

Cugola *et al.* (1997) font la distinction entre deux types de mobilité :

- *la mobilité au sens faible* qui permet à une application d'envoyer du code à un site distant pour le faire exécuter. Le code ainsi transféré peut être accompagné de données d'initialisation, mais il n'y a pas de migration de

variables d'exécution (pile, variables non statiques). C'est le cas par exemple du téléchargement d'une applet Java à partir d'un butineur.

- *la mobilité au sens fort* où le programme est déplacé dynamiquement en cours d'exécution. Dans ce cas, le code et les variables d'exécution sont envoyés au nouvel hôte.

En général, les chercheurs (Greenberg *et al.*, 1998; Oppliger, 1999) identifient deux types de menaces : les attaques d'un agent contre la plate-forme, et réciproquement les attaques de la plate-forme contre un agent. Jansen et Kaygiannis (1999) rajoutent les attaques d'un agent contre un autre agent et celles d'autres entités attaquant un agent ou la plate-forme. Dans cette catégorie, ils incluent les attaques d'un agent contre un agent d'une autre plate-forme et celle d'une plate-forme contre une autre plate-forme.

Attaques agent contre plate-forme

Les attaques agent vers plate-forme peuvent prendre différentes formes. On distingue principalement sept types d'attaques :

- la mascarade
- le déni de service
- l'accès non autorisé, le vol d'information
- les dégâts
- le harcèlement
- l'ingénierie sociale
- la bombe logique.

Une *mascarade* a lieu quand un agent non autorisé prétend être un autre agent. Elle peut avoir pour but d'obtenir des ressources auxquelles l'attaquant n'a normalement pas accès ou de faire endosser la responsabilité de certaines actions à un autre agent.

Un *déni de service* a lieu lorsqu'un agent consomme trop de ressources (UCT, bande passante de la connexion au réseau). Ces attaques peuvent être intentionnelles ou

non (erreur de programmation). Le problème avec un agent mobile est que le code est en principe écrit en dehors de la plate-forme qui l'exécute et il peut contenir du code malveillant.

Quand un agent arrive sur une plate-forme, il faut qu'il soit authentifié puis soumis à la politique de sécurité le concernant. Cela permet d'empêcher l'accès d'utilisateurs ou de processus non autorisés à des ressources protégées. S'ils arrivent à y avoir accès, il y aura *vol d'information*.

Des *dégâts* peuvent être causés lorsqu'un agent détruit ou modifie des ressources ou des services en les reconfigurant, en les modifiant ou en les effaçant de la mémoire ou du disque. Tous les autres agents sur la plate-forme qui utiliseront ce service ou cette ressource seront touchés.

Il y a *harcèlement* lorsqu'un agent mobile ennuie les gens par des attaques répétées : cela arrive lorsqu'un agent mobile montre à répétition des images publicitaires non demandées à l'écran de l'hôte sur lequel il s'exécute.

L'*ingénierie sociale* a lieu lorsque les personnes ou les hôtes sont manipulés par l'agent mobile en utilisant la désinformation ou la coercition. Par exemple, un agent mobile peut demander le mot de passe de l'utilisateur sous la fausse autorité de l'administrateur système.

Une *bombe logique* est une des attaques décrites ci-dessus dont le déclenchement est basé sur un événement externe (une date, un emplacement de l'hôte dans un certain périmètre du réseau).

Attaques plate-forme contre agent

On distingue cinq types d'attaques de la plate-forme vers l'agent :

- la mascarade
- le déni de service
- le vol d'information
- l'altération
- la bombe logique.

Une *mascarade* a lieu lorsqu'une plate-forme se fait passer pour une autre plate-forme dans le but de tromper l'agent par rapport à sa destination normale et de le faire quitter son domaine de sécurité. Cela peut permettre à la plate-forme d'obtenir de l'information sensible contenue dans l'agent. Cela fait du dommage à l'agent et à la plate-forme dont l'identité a été abusée.

Lors d'un *déni de service*, la plate-forme refuse d'effectuer les services demandés par l'agent. Ainsi, elle peut ignorer les demandes de service, introduire des délais inacceptables pour des tâches critiques, ne pas exécuter le code de l'agent ou bien le terminer sans avertissement. D'autres agents qui attendent la réponse de cet agent seront en interblocage (deadlock). Par exemple, si une plate-forme ne donne pas d'accès réseau à un agent mobile, il ne pourra pas se déplacer.

Le *vol d'information* est un danger très important car la plate-forme a accès au code, à l'état et aux variables de l'agent mobile; il est donc dangereux de faire exécuter des algorithmes propriétaires ou de stocker des informations sensibles (clé privée du propriétaire) dans l'agent. L'encryptage classique n'est valable ici que pour le code et les données qui ne seront pas utilisés sur cette plate-forme. En outre, même si la plate-forme n'est pas capable d'obtenir directement de l'information par l'agent, elle peut en déduire par les services demandés, la destination de l'agent ou encore à partir des agents avec qui il communique. Ainsi, si l'agent mobile communique avec un concessionnaire de voiture, la plate-forme pourra avertir un concessionnaire d'une marque concurrente qui démarchera l'agent.

Un agent se déplace sur plusieurs plates-formes, dans des domaines de sécurité différents; la plate-forme a accès au code, à l'état et aux variables de l'agent mobile. En conséquence, il est possible qu'une des plates-formes modifie le code, l'état ou les variables : cela constitue une *altération* de l'agent. La plate-forme peut aussi modifier la communication de l'agent mobile. Cela serait désastreux dans le cas de transactions financières. Un agent prend des risques à chaque fois qu'il se déplace sur une nouvelle plate-forme. De plus, il peut devenir impossible de retrouver la plate-forme responsable

d'une altération si elle n'est pas détectée tout de suite. On peut distinguer deux cas : celui où l'agent mobile se déplace depuis sa plate-forme mère vers une autre puis revient est appelé « single hop problem »; celui où plus d'une plate-forme étrangère sont visitées est appelé « multi-hop problem ». Bien sûr, les risques sont plus importants et plus difficiles à gérer dans le dernier cas.

De même que pour les bombes logiques d'un agent contre une plate-forme, une *bombe logique* d'une plate-forme contre un agent est déclenchée par un événement extérieur. Ainsi, une plate-forme peut attaquer un agent à cause de l'identité de son propriétaire ou des données qu'il transporte.

Attaques autres contre plate-forme

La dernière catégorie d'attaques, *autres contre plate-forme,* intègre quatre types d'attaques :

- la mascarade
- l'accès non autorisé
- le déni de service
- le rejeu.

Une *mascarade* a lieu lorsqu'un agent mobile qui demande un service de manière distante tente de se faire passer pour un autre agent. Ainsi, il pourra avoir accès à des ressources ou des services qu'il ne pouvait obtenir avec son identité réelle.

Certains utilisateurs, processus ou agents, peuvent demander à distance des ressources pour lesquelles ils n'ont pas de droit d'accès. Si jamais la plate-forme donne accès à ses ressources, il y aura *accès non autorisé*. Il faut donc protéger la plate-forme contre des attaques de ce type en appliquant une politique d'accès stricte et en mettant à jour le système lorsque des trous de sécurité sont trouvés.

Les services offerts par la plate-forme de manière distante peuvent être rendus inaccessibles par une attaque conventionnelle de *déni de service*. Quant au *rejeu*, il a lieu

lorsqu'une plate-forme ou un intermédiaire peut copier un agent ou un message et rejouer ce message (intéressant pour un message du type « acheter ») ou relancer l'agent.

Pour parer aux attaques que nous avons vues, il faut protéger à la fois la plate-forme et l'agent. Nous allons passer en revue, dans les prochaines sections, les techniques proposées pour protéger la plate-forme puis celles pour protéger l'agent.

2.2 Protection de la plate-forme

Beaucoup de techniques utilisées pour protéger la plate-forme ou l'agent sont proches de celles utilisées dans une architecture distribuée classique (exemple : client-serveur). D'autres sont tirées de techniques de sécurité du code mobile et du contenu exécutable. Contrairement aux premières, ces dernières ne n'ont pas encore été suffisamment éprouvées.

Le concept d'agent mobile pose des problèmes qui n'ont pas été rencontrés auparavant. Chess (1998) cherche les éléments qui font différer la sécurité des systèmes conventionnels de ceux des agents mobiles. Il y a d'après lui quatre hypothèses que les agents mobiles ne respectent pas :

- lorsqu'un programme effectue une action sur un système conventionnel, il est facile d'identifier la personne à qui cette action peut être attribuée ou qui a effectué cette action;
- tous les logiciels proviennent de sources facilement identifiables et généralement de confiance;
- la plupart des menaces proviennent d'attaquants qui exécutent des programmes dans le but d'obtenir des résultats non autorisés;
- les programmes ne se transmettent que si les personnes le font intentionnellement.

Le but de la protection de la plate-forme est d'empêcher l'agent d'interférer avec celle-ci. La plupart des techniques associées à cette protection sont basées sur le modèle de forteresse selon lequel on cherche à protéger l'hôte en ayant un système fermé qui n'est accessible que par des interfaces bien définies. Dans ce contexte, on utilise le

concept de *moniteur de référence* qui est un programme contrôlant l'accès de l'agent mobile aux informations, ressources et services de la plate-forme. Ce moniteur de référence applique un certain nombre de règles pour l'accès de l'agent à ces ressources. Ces règles sont regroupées dans la *politique de sécurité* de la plate-forme. L'implémentation du moniteur est supposée être toujours appelée (i.e., il est impossible d'obtenir une ressource sans l'appeler), être résistante aux altérations, être suffisamment petite pour être analysée et testée (Jansen et Kaygiannis, 1999).

Les techniques de protection de la plate-forme sont :

- l'authentification des agents (Greenberg et Byington, 1998),
- le carré de sable,
- le contrôle d'accès,
- la vérification du code,
- l'estimation de l'état (Farmer, Guttman et Swarup, 1996),
- l'historique des hôtes (Ordille, 1996),
- le code avec preuve (Necula et Lee, 1998),
- les techniques de limitations (Greenberg et Byington, 1998),
- le journal d'audit.

Authentification des agents

Le problème de l'authentification des agents mobiles est similaire à celui qui se pose en milieu distribué. Ici, on a deux buts : vérifier l'intégrité du code et authentifier ses auteurs et utilisateurs.

Greenberg et Byington (1998) proposent de signer l'agent mobile numériquement avec un algorithme cryptographique à clé publique. Cet algorithme peut soit générer un certificat qui assure l'intégrité du code et qui permet d'authentifier le propriétaire et le producteur de l'AM pour l'hôte, soit crypter l'agent de sorte que seul le réceptionnaire puisse le décoder. Cela assure la confidentialité de l'agent mobile.

Berkovits, Guttman et Swarup (1998) proposent un modèle pour l'authentification des AM. Ils distinguent cinq sujets : l'auteur de l'agent, le programme (écrit et signé par l'auteur), l'expéditeur de l'agent, l'agent (constitué des données et du programme) et les hôtes où il a été exécuté. Il découpe la vie de l'agent en trois périodes : la création du programme, la création de l'agent et la migration de l'agent.

La création du programme consiste en l'assemblage du code et d'une fonction d'estimation de l'état (*state appraisal function*). Cette fonction donne le nombre maximum de permissions qui doit être accordé à l'agent en fonction de son état. Le tout est signé par les auteurs et éventuellement par un organisme de vérification du code. Pour ce qui est de la création de l'agent, l'expéditeur rajoute une autre fonction d'estimation de l'état qui calcule les permissions dont l'agent mobile a besoin pour s'exécuter en fonction de son état. Cela est signé par le créateur. En ce qui concerne la migration de l'agent, il en existe de plusieurs types qui sont basés sur des relations de confiance différentes :

- le changement d'hôte où l'hôte I1 envoie l'agent vers un autre hôte I2;
- la délégation où l'hôte I1 envoie l'agent vers un autre hôte I2 où il sera exécuté sous la responsabilité de l'expéditeur et de I1;
- le changement d'agent quand un agent décide lui-même de migrer, il s'exécute alors sous la responsabilité de son créateur;
- la délégation d'agent où l'agent se délègue à un hôte.

Quand un hôte reçoit une demande pour exécuter un agent, il vérifie la signature de l'auteur du programme et la signature de l'expéditeur. Puis, il authentifie l'agent comme admissible en vérifiant un certain nombre de théorèmes.

Chess (1998) évoque les limites de cette approche qui sont les mêmes que dans une architecture répartie, mais avec encore plus de problèmes. La première limite est bien sûr la distribution des clés publiques car les solutions qui fonctionnent avec un nombre réduit d'utilisateurs ou qui n'utilisent la cryptographie qu'occasionnellement ne sont pas adaptées pour les agents mobiles. Il est important d'avoir un moyen d'obtenir facilement le certificat de la clé publique d'un sujet.

La seconde limite, aussi décrite par Greenberg *et al.* (1998), est le type de certification : on sait qui est l'auteur du programme, mais cela ne veut pas dire que le programme est inoffensif. Pour éviter cela, il faut que des autorités certifient ce caractère inoffensif. Jansen et Kaygannis (1999) ajoutent qu'il est parfois difficile d'avoir des vérifications de qualité en peu de temps. Le problème devient alors un de certification de l'état de l'agent mobile car il a été exécuté sur des hôtes potentiellement dangereux.

Carré de sable

La technique du carré de sable (sandbox) consiste à isoler un programme dans son propre espace de fautes de façon logicielle. L'environnement est restreint en termes de privilèges et de ressources. Le code est exécuté dans une sorte de « carré de sable ». Un agent ne pourra pas modifier la plate-forme, ni les autres agents qui roulent sur la plate-forme. Par exemple, une applet java ne peut pas accéder au système de fichier local.

Contrôle d'accès et d'autorisation

Cette technique se rapproche du « carré de sable ». Le moniteur de référence donne des autorisations à un agent mobile pour un certain nombre de ressources en fonction du résultat de son authentification. Pour savoir quelles autorisations donner à quel agent, le moniteur de référence applique les règles de la politique de sécurité. Ces règles touchent toutes les ressources qu'un agent peut avoir besoin : accès au réseau, aux disques, etc. Dans le langage Java (Gosling et Mc Gilton, 1996), le moniteur de référence est le « security manager », il prend en charge tous les accès aux ressources.

Par exemple, un agent mobile qui tente de lire un fichier dont il ne détient pas les droits de lecture sera arrêté par le moniteur de référence. Cette technique est différente du « carré de sable » car elle permet, d'une part, d'avoir une granularité plus fine sur la protection et l'autorisation et, d'autre part, une adaptation des droits d'accès en fonction des agents. Le problème du carré de sable ne se pose plus car une application avec des

besoins en ressources forts ou particuliers pourra les avoir si la politique de sécurité le permet.

Vérification du code

Le but de cette technique est de vérifier si le code d'un agent mobile est un programme valide. Si un vérificateur de code trouve des opérations illégales dans un agent, celui-ci ne sera pas exécuté. Un exemple de refus pourrait être un programme qui exécute une chaîne de caractères aléatoire comme un segment de programme.

Dans les langages interprétés, cette technique est parfois nommée « interprétation du code inoffensif ». Dans Java, il existe un vérificateur de byte-code qui effectue une vérification sur les types, il fait aussi des vérifications pendant l'exécution (Gosling et Mc Gilton, 1996).

Estimation de l'état

La technique d'estimation de l'état (State Appraisal) est utilisée pour protéger une plate-forme par rapport à un agent dont l'état aurait été modifié par une plate-forme défaillante ou ennemie. À ce titre, elle pourrait aussi apparaître dans les techniques de protection de l'agent, car elle permet de détecter une altération de son état. L'auteur et l'utilisateur de l'agent seront protégés contre une utilisation frauduleuse de leur agent.

Farmer, Guttman et Swarup (1996) ont proposé l'idée de fonction d'estimation de l'état qui prend comme entrée l'état de l'agent et qui donne en sortie un ensemble de ressources dont l'agent a besoin pour exécuter sa tâche.

Le producteur de l'agent et l'utilisateur créent leurs propres fonctions. Chacune est signée numériquement pour empêcher toute modification puis est liée à l'agent. L'auteur écrit une fonction *max* qui donne le maximum de ressources que l'agent peut demander. L'utilisateur de l'agent écrit une fonction *req* qui donne les ressources que demande l'agent; cette fonction peut minimiser le coût que l'utilisateur devra payer pour les ressources de son agent. S'il n'y a pas de modifications de l'état de l'agent, on aura *req(E)* inclus dans *max(E)*. Si jamais ce n'est pas le cas, c'est qu'il y a eu modification

de l'état de l'agent mobile. Ces fonctions permettent aussi à la plate-forme de savoir de quelles ressources l'agent a besoin et de les lui donner si cela est conforme à sa politique de sécurité.

Cependant, la manière dont cette théorie sera mise en pratique n'est pas claire car l'ensemble des états possibles d'un agent est très grand, et écrire des fonctions d'estimation de l'état sera certainement facile pour détecter les attaques évidentes; pour des attaques plus subtiles, cela apparaît plus compliqué. De plus, il n'est pas toujours possible de distinguer un résultat normal d'une tentative de tromperie.

Historique des hôtes

Westhoff *et al.* (1999) et surtout Ordille (1996) ont proposé l'idée d'évaluer la confiance qu'a un hôte dans un agent à partir de son identité et des plates-formes sur lesquelles il s'est exécuté auparavant. Pour cela, il est nécessaire de mettre à jour un journal avec l'identité de toutes les plates-formes qui ont été visitées par l'agent. Ce journal est protégé cryptographiquement pour éviter toutes tentatives de modifications : à chaque fois qu'un agent mobile migre, la plate-forme sur laquelle il est, signe le journal qui indique son identité et celle de la destination de l'agent mobile. Lorsqu'un agent arrive sur une nouvelle plate-forme, cette dernière décide si elle l'exécute ou non et quelles ressources elle lui accorde en fonction de l'historique des hôtes traversés par l'agent.

Cette technique est efficace dans le sens où, avec la signature numérique, une entrée dans l'historique est non répudiable. On peut donc avoir confiance dans l'historique. Le problème de cette technique est que lorsque le nombre d'hôtes visités augmente, l'historique peut prendre un poids prohibitif. De plus, il n'existe pas encore à notre connaissance d'algorithmes pour évaluer l'historique et en déduire le niveau de confiance et de risque.

Code avec preuve

Necula et Lee (1998) ont proposé le code avec preuve ou PCC (Proof Carrying Code) qui permet à un système de déterminer de manière certaine et instantanée si le code qu'il a reçu d'un autre système est sûr à installer et à exécuter. L'idée est que le producteur du code donne une preuve encodée que le code adhère à la politique de sécurité du consommateur. La preuve est encodée de telle sorte qu'elle puisse être transmise numériquement au consommateur et qu'elle puisse être validée rapidement avec un procédé fiable, simple et automatique. Une fois que le code a été vérifié, il peut être exécuté sans risque.

Le PCC se fait en deux étapes. À la première, le consommateur reçoit le code non vérifié et donne un prédicat sur ce code (de type logique du premier ordre) qui, s'il est vérifié, voudra dire que le programme est conforme à sa politique de sécurité. La deuxième étape est la construction de la preuve, cette partie est souvent difficile à faire. Puis, la preuve est envoyée au consommateur qui la vérifie. Cette partie est rapide à effectuer.

Selon Necula et Lee (1998), les points forts de PCC sont la généralité (il permet d'avoir des politiques de sûreté plus abstraites et plus adaptables à la plate-forme que la protection de la mémoire), l'automaticité du mécanisme de vérification et la confiance qui en découle, l'efficacité (la vérification est rapide), le fait qu'il n'y ait pas besoin de relation de confiance, et la flexibilité (PCC est indépendant du langage de programmation). Par rapport aux autres techniques de protection de la plate-forme (isolation logicielle, langage interprété), les résultats au niveau temps d'exécution sont plus rapides. De plus, PCC permet d'éviter d'exécuter du code qui se termine de façon « brutale » : si on utilise une technique de protection de la plate-forme qui demande une vérification en temps réel du code comme le vérificateur de code Java, le code sera exécuté et se terminera « brutalement ». Cela peut prendre du temps à la plate-forme pour reprendre une exécution normale. Le PCC permet donc de gagner du temps UCT en n'exécutant pas cet agent. Enfin, cette technique est préventive, alors que signer le

code permet d'identifier et d'authentifier le code mais ne permet pas d'éviter l'exécution de code dangereux.

Un premier problème réside dans le dilemme entre l'optimisation du code et la difficulté à construire la preuve, car plus le code est optimisé, plus le prédicat est difficile à prouver. Un autre problème vient des contraintes de temps car, pour pouvoir exercer un contrôle du nombre d'instructions, l'implémentation de l'agent a dû rajouter un compteur d'instructions. Une voie de recherche est de concevoir un outil qui rajoute automatiquement ces instructions.

L'automatisation des PCC est aussi une voie active de recherche. Les résultats des expériences montrent que la taille des PCC est importante et augmente parfois exponentiellement avec le code. Des améliorations sont donc possibles. Des recherches peuvent être faites sur d'autres propriétés que le PCC pourrait prouver.

Techniques de limitations

Greenberg et Byington (1998) proposent trois types de limitations sur l'agent qui peuvent aider à protéger un hôte. Ces limitations doivent être choisies en fonction de la tâche de l'agent car, le cas échéant, elles peuvent l'empêcher de l'effectuer :

- Limitation en temps

  L'AM n'a le droit de rester dans la couche d'exécution qu'un certain temps (relatif ou absolu). Une fois qu'il n'a plus de temps, il est détruit ou revient à sa plate-forme de départ.

- Limitation en destination

  Le nombre de destinations possibles de l'agent mobile est limité. De plus, les migrations ne pourront se faire que vers une liste d'hôtes définie dès la création de l'agent ou dans une certaine partie du réseau. Par exemple, l'agent ne pourra aller que sur les plates-formes d'une certaine organisation.

- Limitation de duplication

  Un agent ne peut être autorisé à migrer qu'un certain nombre de fois.

Journal d'audit

Le journal d'audit enregistre toutes les actions de l'agent. Ainsi, après la découverte d'une action frauduleuse, un agent fraudeur pourra être identifié. Cela peut être utile pour des poursuites judiciaires.

Après avoir décrit les techniques pour protéger la plate-forme, nous allons voir les techniques proposées par les chercheurs pour protéger les agents contre les attaques de plates-formes malveillantes.

2.3 Protection des agents

Alors que les contre-mesures visant à protéger les plates-formes sont largement inspirées des systèmes conventionnels en employant des méthodes préventives, les techniques de protection des agents font plutôt de la détection. Cela est dû au fait que l'agent est complètement dépendant de la plate-forme sur laquelle il s'exécute et ne peut, par lui-même, empêcher une attaque. Par contre, il est possible de la détecter ou de la rendre moins dangereuse.

Le principal problème vient des données et des informations d'état. En effet, supposons que l'utilisateur de l'agent signe l'agent (code + données + informations d'état) et l'envoie sur la première plate-forme. Lors de la prochaine migration, il n'y aura plus de vérification possible sur l'état et les données car la plate-forme les aura modifiés et elle est peut être ennemie. Il est possible d'avoir du code signé; cependant, on ne peut pas de protéger les parties variables de l'agent.

La protection des agents contre la plate-forme est un domaine de recherche très important car les problèmes qui se posent sont différents de ceux rencontrés dans les systèmes traditionnels. Parmi les techniques de protection, on trouve :

- l'enregistrement d'itinéraires avec des agents coopérants (Roth 1998a et 1998b),
- les traces cryptographiques (Vigna, 1998; Hohl, 1999),
- le calcul de fonctions cryptographiques (Sander et Tchudin, 1998),
- la boite noire limitée dans le temps (Hohl, 1998),

- la génération de clés à partir de l'environnement (Riordan et Schneier, 1998),
- la protection de l'agent par tolérance aux fautes (Schneier, 1997),
- l'encapsulation des résultats partiels (Young et Yung, 1997; Yee, 1997),
- le tiers de confiance (Wilhem *et al.*, 1998),
- le masquage du code (Bazzi, 1998).

2.3.1 Enregistrement d'itinéraires avec des agents coopérants

Roth (1998a) propose de donner une tâche à deux agents qui vont parcourir un ensemble de plates-formes et qui ont des itinéraires disjoints, un des deux agents mobiles enregistrant l'itinéraire de l'autre. Avant de migrer, un agent envoie à l'autre agent, à travers un canal authentifié, l'identité de la dernière plate-forme, celle de la plate-forme actuelle, ainsi que celle de la plate-forme où il va. L'autre agent maintient un journal de l'itinéraire et vérifie qu'il n'y a pas de contradictions.

Roth (1998a) fait plusieurs hypothèses. La première est qu'aucun hôte sur l'itinéraire d'un agent ne coopère avec un hôte de l'itinéraire de l'autre agent. Il suppose aussi l'existence d'un canal de communications. Enfin, il suppose que les identités sont données sans altération à l'agent.

Toutefois, la méthode recèle un certain nombre d'inconvénients. En effet, mettre en place le canal de communications à chaque migration est une opération coûteuse. Si un agent est tué, le protocole n'est pas capable de savoir lequel des deux hôtes est responsable. Si une plate-forme déclare avoir reçu un agent d'une plate-forme et que cela n'est pas vrai, le protocole ne peut décider qui est coupable. Enfin, si un hôte reçoit deux agents qui viennent du même hôte, il est possible d'interchanger les agents qui enregistrent l'itinéraire des deux autres.

Roth (1998b) adapte un protocole de paiement basé sur le principe de deux agents coopérants. Ce protocole peut être généralisé à plus de deux agents. Pour certaines applications, il serait possible qu'un agent soit statique sur la plate-forme de l'utilisateur.

2.3.2 Traces cryptographiques

Vigna (1998) propose une technique qui permet de détecter toute exécution anormale d'un agent sur une plate-forme à partir d'un fichier, une sorte de « résumé de l'exécution » appelé *trace*. Ces traces sont signées par chaque plate-forme, ce qui permet de retrouver la plate-forme coupable d'une mauvaise exécution de l'agent. À la fin de l'exécution de l'agent, si l'utilisateur de l'agent a des suspicions quant à la bonne exécution de l'agent, il « réexécute » l'agent grâce aux traces et, en cas d'erreur, il peut ainsi retracer la plate-forme coupable.

Une trace est une paire (n,s) où n désigne un identifiant unique d'une ligne de code et s la signature de cette ligne de code. Pour les opérations qui ne modifient les variables que par rapport à d'autres variables internes (opérations blanches), la signature est vide. Le Tableau 2.1 illustre la notion de trace.

Tableau 2.1 Exemple de trace

Code	Trace
10 read(x)	(10, x = 5)
11 y = x+z	(11,-)
12 m = y+1	(12,-)
13 k = cryptInput	(13, k = 2)
14 m = m+k	(14,-)

Après exécution, une plate-forme crée un fichier en appliquant une fonction de hachage aux traces de l'exécution. Ce fichier est ensuite signé par la plate-forme grâce à sa clé privée. Puis, elle envoie ce fichier au prochain hôte de l'agent avec le code et l'état de l'agent. Une fois que l'agent a terminé son exécution et est rentré sur la plate-forme de son propriétaire, ce dernier peut décider s'il veut vérifier l'exécution de l'agent. Si c'est le cas, il réexécute l'agent à partir de son état initial, puis compare le résultat de la fonction de hachage de ses traces avec celui de l'exécution à distance. S'ils sont égaux, c'est que l'hôte n'a pas triché et on passe à l'hôte suivant. Cette approche

détecte toutes les attaques qui donnent un état différent, du moment que l'hôte ne ment pas sur les entrées.

Cependant, des critiques peuvent être émises. La première est celle de la consommation de ressources : les algorithmes à clé publique sont particulièrement consommateurs de temps UCT, la taille des traces peut devenir vraiment importante (même en appliquant plusieurs mécanismes pour en diminuer la taille). De plus, il était supposé que le code était statique. Or, la plupart des langages de type Script proposent le code dynamique. Une autre limite est le fait qu'il faille attendre la fin de l'exécution de l'agent pour pouvoir détecter une attaque. Si une attaque doit être détectée avant un certain temps, cela n'est pas possible. Enfin, il a été supposé que les agents ne partageaient pas de mémoire et n'étaient pas en « multithread ». Une amélioration à ce modèle pourrait être apportée pour ne plus avoir ces limitations.

Hohl (1999, 2000) propose une adaptation de la technique des traces cryptographiques en faisant la vérification de l'exécution à partir des traces, lors de l'arrivée d'un agent mobile sur une nouvelle plate-forme. Il a montré grâce à des mesures que la vérification engendrait une consommation d'UCT à peu près égale à celle de l'exécution. Cette approche permet d'éliminer un certain nombre de désavantages de la méthode de Vigna. En effet, la taille des traces est réduite car on n'échange que les traces d'une exécution d'un hôte et non pas la somme des traces de tous les hôtes visités précédemment; la détection d'une attaque se fait juste après l'attaque. Cependant, des optimisations de la taille des variables d'états (donnée et variables d'exécution) sont possibles. En effet, après son exécution sur une plate-forme, l'agent va être envoyé sur la prochaine plate-forme avec les variables d'états avant exécution et après exécution, ainsi que les traces. Cela permet à la plate-forme suivante de vérifier la bonne exécution de l'agent sur la plate-forme précédente. L'optimisation de la taille des variables d'états peut se faire avec une méthode qui permettrait de choisir comment envoyer ces deux états (avant et après exécution) : soit deux états distincts, soit un état et une différence par rapport à cet état. De plus, une méthode de choix entre une entrée entière fournie par la plate-forme ou le résultat de la première opération sur cette

entrée est souhaitable. En effet, dans la trace, on conserve normalement le résultat de la première ligne de code qui utilise l'entrée. Mais, si l'entrée est de taille inférieure à ce résultat, il est préférable de la conserver. Enfin, il reste le problème des entrées qui peuvent être manipulées par l'hôte sans protection, et celui de deux hôtes ennemis collaborant et qui se suivent dans l'itinéraire de l'hôte.

2.3.3 Calcul de fonctions cryptographiques

Sander et Tschudin (1998) ont proposé une technique pour que l'agent mobile devienne une boite noire pour la plate-forme sur laquelle il s'exécute. Ainsi, ils proposent une méthode pour que l'agent signe des documents. Il s'agit de faire exécuter à la plate-forme un programme qui contient une fonction cryptée sans qu'elle soit capable de décoder cette fonction.

Le problème est le suivant :

«Alice a un algorithme qui calcule la fonction f. Bob a une entrée x et veut calculer f(x) pour Alice, mais elle ne veut pas que Bob apprenne quoique ce soit de f. De plus, Alice et Bob ne doivent pas s'échanger de message pendant le calcul de f(x).»

Pour résoudre ce problème, il faut supposer que A puisse crypter f, ce qui donne E(f). Voilà leur méthode :

- Alice crypte f.
- Alice crée un programme P(E(f)) qui implémente E(f).
- Alice envoie P(E(f)) à Bob.
- Bob applique P(E(f)) à x.
- Bob envoie P(E(f))(x) à Alice.
- Alice décrypte P(E(f))(x) et obtient f(x).

Le problème est maintenant de trouver des modèles cryptographiques qui permettent de calculer E(f). Sander et Tschudin ont trouvé une méthode pour les fonctions f de type polynomial et rationnel pour calculer E(f). Cependant, plus de

recherche est nécessaire dans ce domaine car tous les programmes ne peuvent être représentés comme une fonction polynomiale ou rationnelle. Cette méthode est utilisable pour l'instant dans un domaine très restreint.

2.3.4 Boite noire limitée dans le temps

Hohl (1998) décrit une méthode de type boîte noire pour protéger un agent contre une plate-forme ennemie. Elle consiste à modifier le code de l'agent de telle sorte qu'il fasse le même travail, mais que la lecture du code ne permette pas de comprendre ce que fait la fonction ou de la modifier sans détection. Le problème est qu'il n'existe pas de solution complète à ce problème. Le calcul de fonctions cryptographiques est une solution mais elle n'est pas suffisante car elle ne peut pas être appliquée à tous les agents. C'est pourquoi Hohl introduit le concept de boîte noire limitée dans le temps. Les propriétés sont les mêmes que celles de la boîte noire, mais elles ne sont valables que pour un certain temps qui est lié à l'agent. Dans ce cas, il faut tout d'abord que le code de l'agent change à chaque début de l'intervalle de protection. Il faut aussi des algorithmes de confusion (obfuscating ou mess-up algorithms) qui donnent une analyse du code difficile (donc longue) mais pas impossible pour un attaquant. Hohl décrit les caractéristiques que devront avoir ces algorithmes : l'algorithme devra être paramétrable avec un grand nombre de valeurs possibles, il ne devra pas être possible de casser la protection avant l'exécution.

Le premier problème est la quantification de l'intervalle de protection que donne l'algorithme de confusion : comment déterminer l'intervalle à partir de l'algorithme ? Il manque un modèle qui donne le degré de masquage d'une variable et la complexité pour la retrouver. De plus, il faut déterminer à partir de quel intervalle cette protection est utile pour l'agent : les agents ont besoin de plus ou moins de temps pour effectuer leur travail. Donc, si l'intervalle est trop faible, le nombre d'agents où cette protection pourra être utilisée sera réduit.

2.3.5 Génération de clés à partir de l'environnement

Riordan et Schneier (1998) proposent une méthode pour qu'un agent fasse une certaine action quand une condition de l'environnement est vraie. Une clé est générée quand cette condition est vraie. Elle peut être utilisée pour décrypter du code de l'agent, pour décrypter un message destiné à l'agent dont l'expéditeur ne veut qu'il soit déchiffrable que dans certaines conditions. La plate-forme ne doit pas savoir de quoi dépend cette condition car elle maîtrise tout l'environnement.

Un « agent sans indice » est un agent qui possède du code encrypté et une fonction qui permet de générer la clé de décryptage en fonction de l'environnement. Si l'environnement n'a pas les conditions pour générer la clé, il est impossible de deviner la fonction de l'agent.

Il est possible de générer une clé à partir de données fixes sur un canal (ex : serveur de nouvelles, pages web). Cependant, l'utilité de cette méthode est fonction des entités qui ont accès directement ou indirectement au canal, de celles qui peuvent manipuler les données et de la manière dont varie l'observation du canal en fonction des observateurs.

Il est aussi possible de générer une clé à partir du temps : certaines méthodes permettent de générer la clé seulement avant une certaine date, alors que d'autres ne le permettent qu'après une date. En utilisant les deux, on peut arriver à générer une clé seulement pendant un intervalle de temps fixé.

2.3.6 Protection de l'agent par la tolérance aux fautes

Schneider (1997) propose une méthode pour qu'un agent mobile soit tolérant aux fautes. Une plate-forme fautive peut se comporter comme une plate-forme ennemie. Donc, en appliquant des méthodes pour se protéger des fautes, on peut aider à protéger l'agent contre les attaques de plates-formes. La méthode décrite consiste à utiliser la duplication et le vote. Plusieurs copies du même agent sont envoyées sur plusieurs plates-formes. Chaque plate-forme de l'étape i prend comme entrée la majorité des entrées qu'elle a reçues des plates-formes de l'étape précédente (i –1). Quand elle a fini,

elle envoie son résultat à toutes les plates-formes de l'étape i+1. Il est aussi possible de savoir quelle plate-forme a donné des résultats erronés grâce à un historique des chemins.

Cette technique s'applique aux applications où les agents peuvent être dupliqués, où la tâche peut être décomposée sans problème et où la survie est une question importante. Cependant, l'inconvénient majeur est, bien sûr, les ressources additionnelles qui sont consommées par la duplication. Un problème possible est de vérifier qu'un des prédicats ne divergent pas car, dans ce cas, ce prédicat n'enverrait pas de résultats aux prochaines plates-formes. En effet, la technique utilisant les machines à états n'est pas applicable ici car les réplicats peuvent exécuter des requêtes différentes ou des requêtes dans un ordre différent.

2.3.7 Encapsulation des résultats partiels

Cette méthode consiste à détecter une altération de l'agent en encapsulant les résultats des calculs de l'agent à chaque plate-forme visitée. La vérification de la validité de ces résultats se fait sur la plate-forme du propriétaire ou sur un intermédiaire placé sur la trajectoire de l'agent. L'encapsulation a plusieurs sous-buts comme la confidentialité (assurée par la cryptographie), l'intégrité et la non-répudiation (assurées par une signature numérique). Les informations qui sont encapsulées dépendent de l'agent. Il y a trois manières d'encapsuler les résultats : avec l'agent, avec la plate-forme ou avec un troisième parti de confiance. La dernière solution n'a pas été véritablement explorée, mais a seulement été évoquée par Yee (1997).

Young et Yung (1997) ont proposé une méthode cryptographique basée sur la cryptographie à clé publique permettant d'encrypter de petits volumes de données en résultats. L'intérêt est que le volume des données encryptées est petit. Cela s'adapte bien au cas des agents mobiles car les résultats partiels sont petits et, avec sa capacité de stockage limitée, l'agent ne peut pas transporter de grands volumes de données cryptées. Les deux auteurs ont montré non formellement comment modifier ce schéma

d'encryptage pour que la corrélation entre des données encryptées soient plus difficilement calculable. Cela permet de rendre les agents moins traçables.

Yee (1997) propose d'encapsuler les résultats dans un PRAC (code d'authentification des résultats partiels). Avant de migrer vers une autre plate-forme, l'agent calcule un « résumé » (le PRAC) des résultats partiels. La clé utilisée est choisie parmi une liste de clés secrètes que l'agent et son propriétaire connaissent. Une fois le calcul effectué, la clé est effacée. Un calcul de code d'intégrité de message est effectué pour assurer l'intégrité des résultats partiels. Puis, l'agent migre vers la nouvelle plate-forme en ayant avec lui le résultat partiel et le PRAC. Si une des plates-formes suivante est ennemie, elle ne pourra pas modifier le résultat partiel sans détection car elle ne pourra pas calculer le PRAC : elle n'a pas la clé associée.

Jansen et Kaygannis (1999) ont mis en évidence plusieurs limitations à cette technique. En effet, si une plate-forme conserve l'ensemble de clés ou la fonction génératrice de clé et que l'agent revisite cette plate-forme ou une plate-forme alliée à cette dernière, un résultat partiel pourra être modifié sans possibilité de détection. De plus, la méthode ne permet pas d'assurer la confidentialité des résultats. Pour y remédier, on peut ajouter une étape de cryptage des données.

Roth (1998) présente une méthode orientée plate-forme qui est une amélioration du PRAC. Chaque plate-forme rajoute ces résultats à la chaîne de résultats en les liant aux précédents : une plate-forme signe numériquement ses résultats avec sa clé privée, et applique une fonction de hachage à la concaténation de la chaîne et de ses résultats cryptés. Cette méthode assure l'intégrité en cryptant chaque résultat avec la clé du propriétaire de l'agent. Cette intégrité est plus forte que le PRAC car il est impossible pour une plate-forme de modifier sa propre entrée si elle est revisitée par l'agent mobile, ou pour une plate-forme collaboratrice.

2.3.8 Tiers de confiance

Wilhem *et al.* (1998) étudie la notion de confiance dans le cadre des agents mobiles. La confiance dans un tiers est définie comme la croyance qu'il va respecter sa

politique de sécurité. Les auteurs discernent deux types d'approches : une optimiste et une pessimiste. Dans l'optimiste, on fait a priori confiance à l'autre entité et on essaye de détecter les violations de politique. Si jamais c'est le cas, il y a une sentence. On distingue alors deux types de confiance : celle basée sur la réputation et celle basée sur une sentence explicite. Dans l'approche pessimiste, on prévient les violations de la politique de sécurité.

L'approche de Wilhem *et al.* (1998) consiste à déléguer la confiance dans un matériel résistant aux altérations, qui fournit à l'agent mobile un environnement d'exécution sûr. Cela permet aux données d'être protégées de manière sure. Le problème est la protection de cet environnement car tout système est faillible pourvu que l'attaquant ait suffisamment de ressources et de temps.

2.3.9 Masquage du code

Bazzi (1998) propose une définition pour le masquage du code et pour la résistance aux modifications. Il décrit sa méthode qui consiste à rajouter du code de manière systématique et automatique en gardant la même fonction. Une proposition de solution pour les fonctions polynomiales est faite, le masquage et la résistance aux modifications sont prouvés. Il reste à mieux spécifier le problème du masquage du code en tenant compte notamment de la quantité de connaissances qu'a l'hôte sur le code. Il est aussi important de définir une évaluation des algorithmes de masquage.

2.4 Synthèse des problèmes ouverts

Au sujet de la protection de la plate-forme, beaucoup a été fait. En particulier, le code avec preuve constitue une idée intéressante qui peut être améliorée à deux niveaux : il faut optimiser la méthode pour diminuer la taille de la preuve et trouver de nouvelles propriétés qu'elle peut prouver. Par ailleurs, l'estimation de l'état mériterait une tentative de mise en pratique car, si l'idée en soi est bonne, la mise en pratique réelle n'a pas été décrite jusqu'ici.

Tout compte fait, les résultats obtenus en matière de protection de la plate-forme ne sont que dans un domaine très restreint. Ainsi, la boite noire limitée dans le temps n'a donné que des algorithmes de confusion (mess-up algorithm) très simples, le calcul de fonctions cryptographiques est aussi possible mais uniquement pour les fonctions polynomiales et rationnelles. Dans ces deux cas, il serait intéressant de trouver des algorithmes applicable à un contexte plus large. Pour la boite noire limitée dans le temps, il faudrait aussi un moyen d'évaluer les algorithmes.

D'autres techniques de protection sont plus applicables. Ainsi, il est possible d'améliorer le protocole de Roth (1998) (enregistrement d'itinéraires avec des agents coopérants) pour ne plus avoir les faiblesses de détection de certaines attaques. Une implémentation serait aussi bienvenue pour évaluer la performance de sa technique. L'utilisation des traces cryptographiques est aussi une idée intéressante que Hohl (1999) a appliquée et qui fonctionne pour détecter un grand nombre d'attaques. Deux questions demeurent cependant en suspens : comment protéger les entrées qui sont fournies par la plate-forme à l'agent et comment améliorer le protocole notamment pour éviter l'attaque de deux hôtes ennemis qui se suivent sur l'itinéraire de l'agent ?

CHAPITRE III
ENREGISTREMENT D'ITINÉRAIRE PAR AGENTS COOPÉRANTS

L'enregistrement d'itinéraire s'effectue selon un protocole dont le but est d'enregistrer et de vérifier l'itinéraire d'un agent mobile par un agent coopérant, de manière autonome, sans avoir une plate-forme de confiance connectée au réseau en permanence. Cela est permis par le concept d'agents coopérants. La plate-forme du propriétaire des deux agents se connecte au réseau pour envoyer les agents sur leur première plate-forme, puis elle peut se déconnecter jusqu'au retour des deux agents. L'itinéraire n'étant pas préétabli, l'agent se déplace au fur et à mesure qu'il accomplit sa tâche, et il choisit les prochaines destinations pendant son exécution sur les différentes plates-formes. Cela peut se justifier dans le cas où l'agent ne connaît que l'adresse de plusieurs bases de données de plates-formes qu'il interroge pour ensuite se déplacer vers les plates-formes qui l'intéressent. Dans ce chapitre, nous allons tout d'abord décrire le protocole original de Roth (1998), puis nous proposerons un protocole inspiré de ce dernier qui permet d'obtenir des résultats meilleurs au niveau de la détection et donc de la protection, ceci avec des hypothèses plus faibles sur les possibilités d'attaques.

3.1 Protocole de Roth

Le but de ce protocole est d'enregistrer et de vérifier l'itinéraire d'un agent mobile dont l'itinéraire n'est pas fixé au départ. Pour expliquer son fonctionnement,

nous allons d'abord énoncer les hypothèses formulées par Roth (1998), puis expliquer le fonctionnement du protocole. Enfin, nous terminerons par la preuve de la sécurité du protocole, en faisant l'inventaire des attaques possibles tout en précisant la manière dont le protocole les détecte et dont les coupables sont désignés.

3.1.1 Définitions et hypothèses

Soit H l'ensemble des plates-formes disponibles sur un réseau.

Soit R un sous-ensemble de $H \times H$ tel que $(h_i , h_j) \in R \Leftrightarrow h_i$ et h_j collaborent dans le but d'attaquer l'agent.

Soit H_a et H_b des sous-ensembles non vides de H avec $(H_a \times H_b) \cap R = \varnothing$. Ces deux sous-ensembles sont dits *non collaborants.*

Deux agents *a* et *b* sont dits *coopérants* lorsque l'itinéraire de l'agent *a* ne contient que des plates-formes de H_a et que l'itinéraire de l'agent *b* ne contient que des plates-formes de H_b. Un hôte spécial est l'hôte d'origine de l'agent qui doit être de confiance. On appelle h_a et h_b les environnements d'exécution en cours respectivement de l'agent *a* et de l'agent *b*. De manière plus simple, l'agent *a* peut être attaqué par la plate-forme qui l'exécute, h_a, mais cette dernière ne peut pas directement attaquer l'agent coopérant *b*. De plus, la plate-forme h_b ne collaborera pas avec h_a dans ce but. Par contre, il est possible que deux plates-formes de l'itinéraire de l'agent *a* collaborent pour attaquer ce dernier. La Figure 3.1 illustre ces hypothèses.

Roth (1998) justifie ce choix sur les itinéraires des agents par le fait qu'il est aussi irréaliste de dire que «toutes les plates-formes sont hostiles et sont prêtes à collaborer entres elles dans le but d'attaquer l'agent», que de dire que «toutes les plates-formes peuvent être utilisées avec confiance». C'est pourquoi, d'après lui, il est plus réaliste de dire que :

- à un moment donné dans la vie de l'agent, il y a un certain pourcentage de plates-formes qui peuvent être considérées comme dangereuses;
- toutes les plates-formes dangereuses ne sont pas prêtes à collaborer avec d'autres plates-formes pour attaquer l'agent.

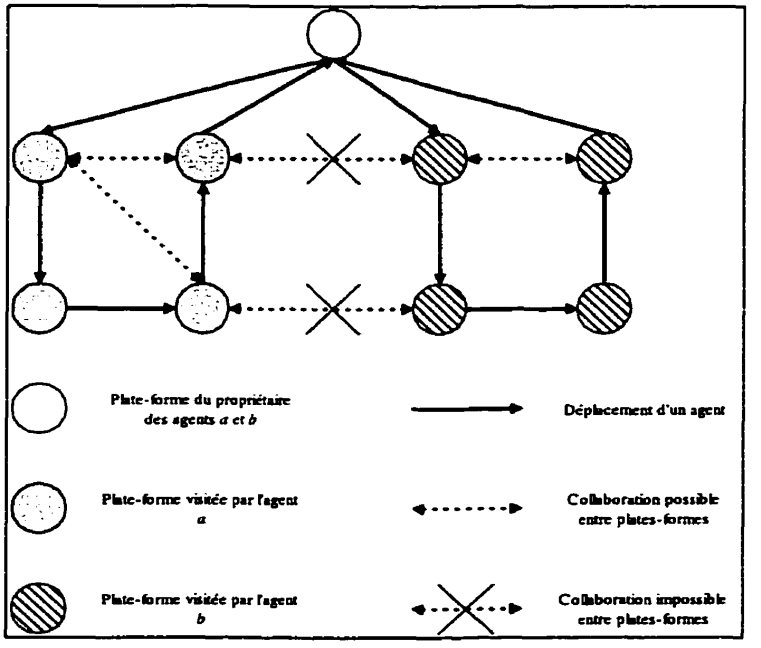

Figure 3.1 Hypothèses de Roth sur les collaborations entre plates-formes

Roth (1998) fait trois hypothèses supplémentaires pour pouvoir appliquer le concept d'agents coopérants à l'enregistrement d'itinéraire. Cependant, il n'en fait pas le développement et ne les a pas implémentées. Ces hypothèses sont les suivantes :

- Le transport des agents d'une plate-forme à l'autre se fait par des canaux authentifiés (Hypothèse 1);
- Les plates-formes fournissent un canal de communication authentifié aux agents coopérants (Hypothèse 2);
- L'agent a accès, de manière authentifiée, à l'identité de la plate-forme distante avec laquelle l'agent communique, à celle de la plate-forme sur laquelle il s'exécute et à celle de la précédente plate-forme où il s'était exécuté (Hypothèse 3).

3.1.2 Le protocole d'enregistrement d'itinéraire

Soient *a* et *b* deux agents coopérants, H_a et H_b deux sous-ensembles de plates-formes de H non collaborantes. Chacun des deux agents doit retourner à sa plate-forme

d'origine à la fin de la tâche. L'agent *b* enregistre et vérifie l'itinéraire de son agent coopérant *a* selon le protocole décrit à la Figure 3.2.

Définition :

Soit $h_i \in H_a$ la $i^{ème}$ plate-forme visitée par l'agent *a* et soit $id(h_i)$ l'identité de cette plate-forme h_i. Soit $prev_i$, l'identité de la dernière plate-forme où il s'est exécuté. Enfin, soit $next_i$ l'identité de la prochaine plate-forme où l'agent désire aller après la plate-forme h_i. L'agent débute sur la plate-forme h_0. Ainsi, si l'agent se déplace n fois, on a $h_0 = h_n$ car, par hypothèse, l'agent *a* revient sur la plate-forme d'origine.

Initialisation :

Soit h_0 la plate-forme d'origine des agents *a* et *b*. h_0 doit être une plate-forme de confiance pour *a* et *b*. Pour les deux agents, $next_0$ a pour valeur l'identité de la plate-forme sur laquelle les agents vont s'exécuter. Puis, chacun des agents est envoyé vers leur première plate-forme par l'intermédiaire d'un canal de communication authentifié (Hypothèse 1).

Étape *i*, $i \in \{1,...,n\}$:

L'agent *a* envoie un message à *b* contenant $next_i$ et $prev_i$ à l'agent *b* au travers du canal authentifié (Hypothèse 2). Ainsi, l'agent *b* apprend $id(h_i)$ (Hypothèse 3). L'agent *b* vérifie que

$$id(h_i) = next_{i-1} \text{ et } prev_i = id(h_{i-1}) \ .$$

Si c'est le cas, il enregistre $next_i$ dans l'itinéraire de l'agent.

Figure 3.2 Protocole d'enregistrement d'itinéraire

3.1.3 Sécurité du protocole

L'attaque qu'on veut détecter correspond à celle où la plate-forme h_i envoie l'agent a vers une plate-forme h_{i+1} telle que $id(h_{i+1}) \neq next_i$. Cela revient à dire que l'agent a a été envoyé vers une plate-forme différente de celle où il désirait aller.

La plate-forme h_i a deux manières d'effectuer cette attaque : elle peut soit modifier la communication entre l'agent a et l'agent b (cette attaque est détectée au retour de l'agent a sur sa plate-forme d'origine), soit ne pas modifier la communication et envoyer l'agent a vers une plate-forme autre que celle où l'agent veut aller (cette attaque est immédiatement détectée).

Attaque avec modification de la communication entre a et b

Comme décrit précédemment, l'agent a envoie un message avec $next_i$ et $prev_i$ à l'agent b, $next_i$ est l'identité de la plate-forme où veut aller l'agent. Comme la plate-forme h_i contrôle l'environnement d'exécution de l'agent, elle peut envoyer à l'agent b un $next_i$ qui correspond à la plate-forme où elle veut envoyer l'agent, et non où l'agent a veut réellement aller. La Figure 3.3 illustre cette attaque.

La vérification des égalités $id(h_i) = next_{i-1}$ et $prev_i = id(h_{i-1})$ est effectuée sans problème à l'étape i, car la modification a touché le $next_i$ qu'a envoyé l'agent a à l'agent b. À l'étape $i+1$, il en est de même car le $next_i$ ainsi modifié correspond à l'identité de la plate-forme où h_i a envoyé l'agent a, la vérification $next_i = id(h_{i+1})$ est effectuée. L'attaque n'est donc pas immédiatement détectée. Cependant, elle le sera au retour de l'agent sur sa plate-forme d'origine. En effet, il est supposé que le propriétaire de l'agent puisse vérifier l'itinéraire de l'agent. Roth n'explique pas comment il est possible que cette vérification soit faite concrètement. Une solution à laquelle nous avons pensé est la suivante : quand l'agent revient sur la plate-forme du propriétaire, ce dernier vérifie la partie fixe de l'itinéraire qui est constituée des adresses des plates-formes connues lors de la création de l'agent. Cette partie fixe comprend au moins une adresse car il faut savoir, au départ, où on envoie l'agent. Puis l'agent ou son propriétaire publie

l'itinéraire sur une page web. Les plates-formes qui ont rajouté des plates-formes à l'itinéraire consultent cette page pour savoir si l'agent a bien visité ces plates-formes. Si une plate-forme s'aperçoit que cela n'est pas le cas, elle avertit le propriétaire, par courrier électronique par exemple, qu'il y a eu une attaque sur l'itinéraire de l'agent et que son résultat n'est pas valable. On est sur qu'une plate-forme ne trichera pas à cette étape car, si elle a rajouté une plate-forme à l'itinéraire de l'agent, c'est qu'elle veut que l'agent s'exécute sur cette plate-forme. On peut supposer que le propriétaire de l'agent donne un certain temps pour que ces plates-formes l'avertissent d'une attaque. Une fois ce temps écoulé, l'utilisateur considère que l'itinéraire est valide et qu'il n'y a pas eu d'attaques sur ce dernier. L'inconvénient de cette solution est qu'elle requiert des interactions supplémentaires d'une part lorsque le propriétaire publie l'itinéraire sur le web et, d'autre part, dans le cas d'une attaque avec modification, il faut envoyer l'avertissement qu'il y a eu une attaque au propriétaire.

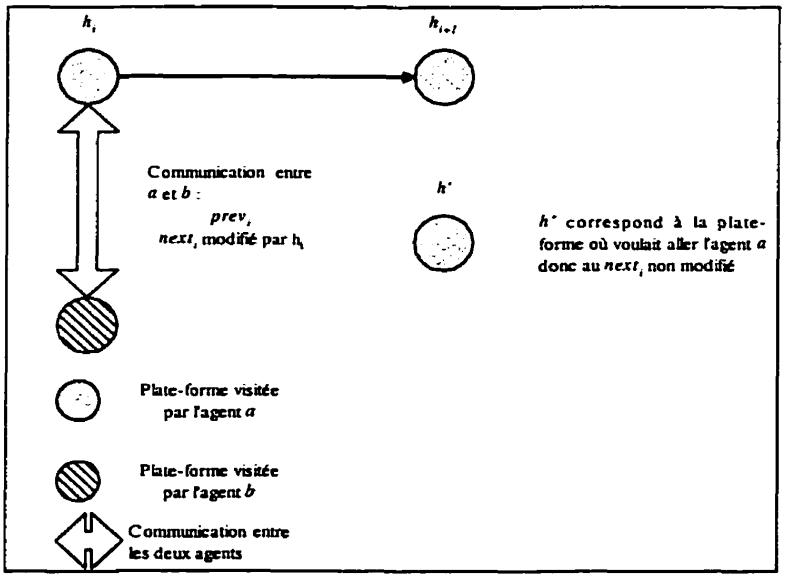

Figure 3.3 Attaque avec modification de la communication entre a et b

De plus, on est sûr que l'enregistrement de l'itinéraire n'a pas été modifié par une plate-forme sur laquelle *b* s'est exécuté pour rendre l'attaque indétectable. Il faut pour cela remplacer l'entrée i+1 de l'enregistrement de l'itinéraire par l'identité de la plate-

forme où voulait aller l'agent. Cela est, par hypothèse, impossible car il est supposé qu'il n'y a pas de collaboration entre les plates-formes visitées par l'agent *a* et celles visitées par l'agent *b*. Or, modifier l'enregistrement de l'itinéraire de cette manière n'est pas profitable pour la plate-forme sur laquelle s'exécute l'agent *b* mais l'est pour la plate-forme h_i. Il est donc impossible que l'enregistrement de l'itinéraire soit modifié.

Attaque sans modification de la communication entre a et b

Si la plate-forme h_i envoie l'agent *a* vers une plate-forme h_{i+1} telle que $id(h_{i+1}) \neq next_i$ et que la communication entre *a* et *b* n'est pas modifiée, alors la plate-forme h_{i+1} devra soit faire une mascarade dans le but d'usurper l'identité de $next_i$, soit empêcher la communication entre les agents coopérants pour que l'attaque ne soit pas détectée. En effet, lors de l'étape i+1, l'agent *b* va vérifier l'égalité $next_i = id(h_{i+1})$ qui ne sera pas exacte, car h_i a envoyé *a* vers une plate-forme autre que celle d'identité $next_i$. La Figure 3.4 illustre cette attaque.

Donc, si la plate-forme h_{i+1} permet la communication entre les agents, alors l'agent *b* s'aperçoit que l'agent *a* n'a pas été envoyé vers la destination voulue et désignera le coupable, h_i. De plus, la plate-forme h_{i+1} ne peut pas remettre l'agent sur sa route en l'envoyant vers l'hôte avec l'identité $next_i$ car l'agent *b* a enregistré $id(h_i)$ comme $prev_i$. Si c'est le cas, la vérification de $prev_i = id\ (h_{i-1})$ engendrera une erreur et le coupable est la plate-forme d'identité $prev_i$.

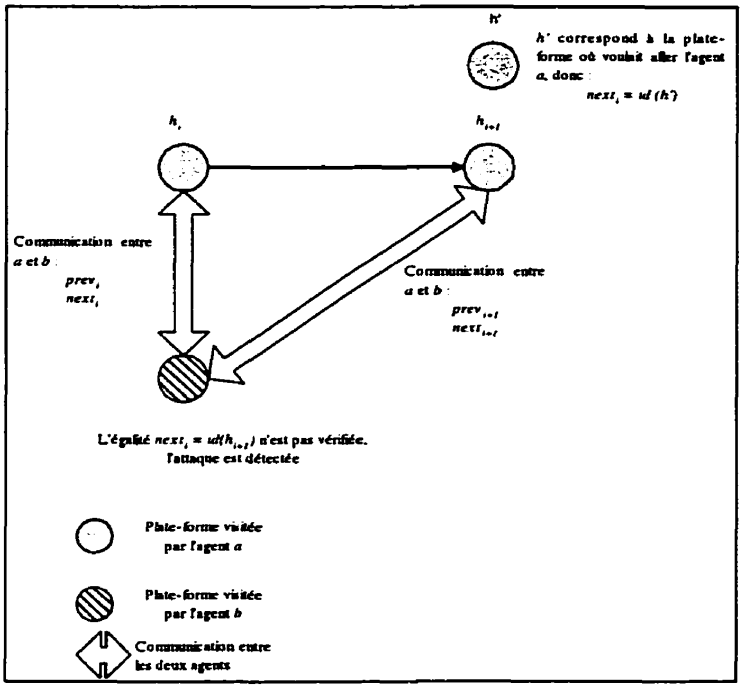

Figure 3.4 Attaque sans modification de la communication entre a et b

En conclusion, h_{i+1} est soit honnête en donnant son identité réelle et *b* voit que h_i est coupable, soit il collabore avec h_i en remettant l'agent sur sa route normale, pour cela il faut que h_i le fasse lui-même ou que ce dernier donne ses clés d'authentification à h_{i+1}. En effet, d'après l'hypothèse 1, le transport des agents se fait de manière authentifiée. Cela revient à ce que les deux plates-formes possèdent une copie de l'agent ou que h_i et h_{i+1} soient rassemblés sous l'identité de h_i. Ces problèmes sont très difficiles à éviter ou à détecter.

Justification du besoin d'un agent coopérant

Le besoin d'un agent coopérant se justifie principalement par le fait que les données que transporte un agent sont susceptibles d'être modifiées par une plate-forme malveillante. En effet, on pourrait argumenter que, si on suppose que la troisième hypothèse est vérifiée, il suffit à l'agent de conserver l'identité authentifiée de chaque plate-forme où il s'est exécuté. On peut aussi supposer qu'il fait les mêmes vérifications que l'agent *b* du protocole de Roth. Cependant, cette solution n'est pas sécuritaire et

voici un contre exemple avec une attaque qui n'est pas détectée. L'hôte h_i attaque l'agent en l'envoyant vers un hôte h' alors que l'agent voulait aller vers la plate-forme h_{i+1}. L'identité de l'hôte h' est ajoutée à l'enregistrement de l'itinéraire. Ensuite, l'hôte h' remet l'agent sur sa route en l'envoyant à h_{i+1} qui est supposé honnête. Sur h_{i+1}, l'agent vérifie l'identité de la plate-forme qui l'a envoyé et cela est correct car h' n'a pas modifié l'enregistrement de l'itinéraire. À l'étape $j > i + 1$, l'hôte h_j, qui est allié avec les plates-formes attaquantes h_i et h', modifie l'enregistrement de l'itinéraire pour enlever l'entrée correspondante à l'exécution sur l'hôte h'. Quand l'agent revient sur la plate-forme de son propriétaire, alors qu'il s'est exécuté sur l'hôte h', ce dernier n'est pas dans l'enregistrement de l'itinéraire.

3.2 Amélioration proposée au protocole de Roth

Comme nous l'avons vu, le protocole de Roth (1998) n'est pas très précis en ce qui a trait aux déplacements de l'agent *b*. Il nous est simplement indiqué qu'il enregistre l'itinéraire de l'agent *a*. Nous proposons de déplacer l'agent *b* à chaque fois que l'agent *a* se déplace, i.e. lorsque *a* contactera *b* pour lui annoncer l'identité de la prochaine plate-forme où il va s'exécuter et celle de la plate-forme précédente qu'il a visitée. Une fois ces informations acquises, l'agent *b* va se déplacer vers une nouvelle plate-forme et attendre que l'agent *a* communique avec lui. De la même manière que nous l'avons fait pour le protocole de Roth, nous allons d'abord expliciter les hypothèses nécessaires à notre protocole, puis son fonctionnement. Ensuite, nous ferons une description des attaques possibles avec nos hypothèses et indiquerons la manière dont le protocole les détecte. Enfin, nous montrerons les apports de notre protocole par rapport à l'original.

3.2.1 Hypothèses

Comme il a été mentionné précédemment, l'agent *b* va se déplacer. On a deux possibilités pour l'itinéraire de *b* : soit il est fixé à l'avance, soit il ne l'est pas. Bien qu'il soit, à première vue, plus logique d'avoir un itinéraire de *b* non fixé au départ, nous allons proposer un itinéraire de *b* fixé au départ des deux agents. Ce choix se justifie en regard des avantages et inconvénients de chacune des deux possibilités. En effet, avec un

itinéraire fixé à l'avance, on simplifie le protocole et on a une meilleure détection des attaques, ceci avec des hypothèses moins restrictives sur les collaborations possibles entre plates-formes.

On suppose que l'agent *a* possède une liste ordonnée des plates-formes sur lesquelles l'agent *b* va aller. On nomme cette liste *pf*. *pf[i]* désigne l'identité de la $i^{ème}$ plate-forme visitée par l'agent *b*, appelée par la suite hb_i. Il se pose alors le problème du nombre de plates-formes qu'on doit mettre dans cette liste. En effet, un protocole d'enregistrement d'itinéraire est intéressant surtout si on enregistre et vérifie l'itinéraire d'un agent se déplaçant sur des plates-formes de manière non prédéfinie. Comme à chaque fois que l'agent *a* se déplace, l'agent *b* se déplace aussi, le nombre de plates-formes visitées par l'agent *b* est égal au nombre de celles visitées par l'agent *a*. Or, on ne connaît pas à l'avance le nombre d'hôtes que va visiter l'agent *a*. Il y a donc un problème pour connaître le nombre de plates-formes qu'on doit mettre dans la liste. Pour solutionner ce problème, on fixe un majorant du nombre de plates-formes visitées par l'agent *a*.

À titre d'exemple, ce genre d'estimation est possible dans des applications du type « recherche de meilleur prix pour un produit », où on sait que l'agent ira voir un certain nombre de marchands, mais on ne sait pas précisément sur quelles plates-formes (un marchand peut avoir plusieurs plates-formes), ni dans quel ordre l'agent va visiter ces plates-formes. Donc, établir un itinéraire avant le départ de la plate-forme du propriétaire est impossible. L'itinéraire de l'agent *b* que consulte l'agent *a* doit bien sûr être protégé contre les modifications. Il peut, par exemple, faire partie des données fixes de l'agent et être protégé contre la modification avec le code de l'agent. Cela est possible avec une signature du propriétaire de l'agent qui permet de s'assurer de l'intégrité de l'agent et de faire connaître cette identité aux plates-formes sur lesquelles vont s'exécuter les agents.

On suppose que l'ensemble des plates-formes visitées par l'agent *a* et celui des plates-formes visitées par l'agent *b* sont *disjoints*.

Le choix de l'ordre des plates-formes visitées par l'agent *b* importe peu. Il est quand même important de minimiser la probabilité d'avoir une attaque de la part d'une plate-forme sur l'itinéraire de l'agent *b* avec une collaboration d'une plate-forme sur l'itinéraire de l'agent *a*. Il est cependant difficile de pouvoir quantifier ce risque de manière à le minimiser, d'autant que l'itinéraire de l'agent *a* n'est pas connu au départ des deux agents. C'est pourquoi on peut choisir l'itinéraire de l'agent *b* de manière aléatoire, parmi une liste de plates-formes dont on sait qu'elles ne proposent pas les services que l'agent *a* recherche. Par exemple, si l'agent *a* a pour but de rechercher le meilleur prix sur un produit, on pourra envoyer l'agent *b* sur des plates-formes qui proposent un service d'exécution de calcul.

La Figure 3.5 illustre ces hypothèses en donnant un exemple de collaboration que Roth supposait impossible dans son protocole et que nous avons rendu possible par notre approche.

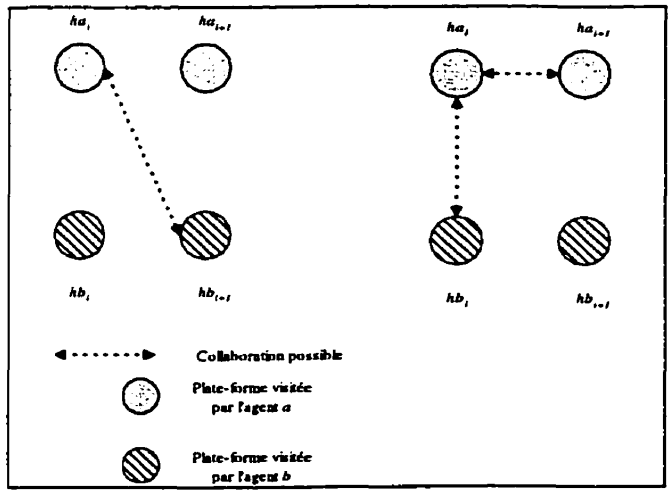

Figure 3.5 Collaborations possibles

On ajoute aussi les mêmes hypothèses que celles du protocole original :

- Le transport des agents d'une plate-forme à l'autre se fait par des canaux authentifiés (Hypothèse 1);

- Les plates-formes fournissent un canal de communication authentifié aux agents coopérants (Hypothèse 2);
- L'agent a accès, de manière authentifiée, à l'identité de la plate-forme distante, à celle de la plate-forme sur laquelle il s'exécute et à celle de la précédente où il s'était exécuté (Hypothèse 3).

3.2.2 Protocole d'enregistrement d'itinéraire

Définition

Soit ha_i la $i^{ème}$ plate-forme visitée par l'agent a et soit $id(ha_i)$ l'identité de cette plate-forme, hb_i la $i^{ème}$ plate-forme visitée par b et l'identité de cette plate-forme $id(hb_i)$. Soit $preva_i$ l'identité de la dernière plate-forme où l'agent a s'est exécuté. D'après l'hypothèse 3, il a accès à cette identité de manière authentifiée. Enfin, soit $nexta_i$ l'identité de la prochaine plate-forme que l'agent désire visiter après la plate-forme ha_i. L'agent débute sur la plate-forme ha_0. Ainsi, si l'agent a se déplace n fois, on a $ha_0 = ha_n$ car, par hypothèse, il revient finalement sur la plate-forme d'origine.

Initialisation

Soit h_0 la plate-forme d'origine des agents a et b. h_0 doit être une plate-forme de confiance pour a et b. Pour l'agent a, $nexta_0$ a pour valeur la plate-forme sur laquelle les agents vont s'exécuter. Chacun des agents est envoyé vers leur destination.

Étape *i*, $i \in \{1,\ldots,n-1\}$

L'agent b attend la communication de l'agent a. Si ce dernier ne communique pas avec lui avant un certain temps, l'agent b considère que l'agent a a été kidnappé et b rentre sur la plate-forme de son propriétaire. Le coupable ne peut alors être désigné de manière certaine : il s'agit, soit de la plate-forme ha_{i-1} qui n'a pas envoyé l'agent, soit de la plate-forme ha_i qui empêche la communication.

L'agent a tente de communiquer avec l'agent b sur la plate-forme hb_i.

Si l'agent *b* n'est pas sur cette plate-forme, c'est que la plate-forme précédente hb_{i-1} ne l'a pas envoyé, que la plate-forme hb_i ne l'exécute pas ou ne lui permet pas d'accéder au réseau. On ne peut donc pas enregistrer ni vérifier l'itinéraire de l'agent *a*. L'agent *a* retourne sur sa plate-forme d'origine.

Sinon, l'agent *a* envoie à l'agent *b* qui est sur la plate-forme hb_i l'identité de $nexta_i$ et celle de $preva_i$, signées par la plate-forme ha_i et, par le canal authentifié, *b* apprend $id\ (ha_i)$.

Il vérifie la signature de ha_i.

Il vérifie les égalités suivantes:

$$id(ha_i) = nexta_{i-1} \text{ et } preva_i = id\ (ha_{i-1})$$

Si ces égalités sont vérifiées, il enregistre $nexta_i$, $preva_i$, ainsi que la signature de ha_i, puis demande à hb_i de signer le tout. Par la suite, on désigne par $sign_{hai}$ (X) le résultat de la signature cryptographique de la plate-forme ha_i sur l'objet X et par || l'opérateur de concaténation.

La Figure 3.6 montre une entrée de l'enregistrement de l'itinéraire. Puis, l'agent *b* se déplace vers la prochaine plate-forme de son itinéraire.

$preva_i$	$nexta_i$	$sign_{hai}(nexta_i \| preva_i)$	$sign_{hbi}\ (preva_i \| nexta_i \| sign_{hai})$

Figure 3.6 Entrée de l'enregistrement de l'itinéraire

Si les égalités ne sont pas vérifiées, c'est qu'il y a eu une attaque. L'agent *b* migre vers la plate-forme de son propriétaire.

L'agent *a* exécute sa tâche sur la plate-forme, puis migre jusqu'à sa prochaine destination. La Figure 3.7 décrit l'algorithme pour l'agent *a*, alors que la Figure 3.8 présente l'algorithme pour l'agent *b*.

Étape n

Lorsque les agents sont revenus sur la plate-forme de leur propriétaire et qu'ils n'ont pas détecté d'attaque, celui-ci vérifie que l'itinéraire n'a pas été modifié. Pour chaque entrée i ∈ {1, ..., n}, il vérifie que la signature $sign_{hai}(preva_i \| nexta_i)$ est valide et que $ha_i = nexta_{i-1} = preva_{i+1}$; il fait de même pour $sign_{hbi}$ ($preva_i \| nexta_i \| sign_{hai}$), hb_i étant la ième entrée dans l'itinéraire. Si une entrée n'est pas valide, c'est qu'il y a eu une attaque sur l'itinéraire de l'agent et qu'on ne peut donc pas faire confiance au résultat que l'agent a trouvé. Cependant, le coupable ne peut pas être trouvé.

Une fois l'intégrité de l'enregistrement de l'itinéraire prouvée, le propriétaire de l'agent vérifie que l'itinéraire de l'agent *a* s'est exécuté sur des plates-formes conformes. Si ce n'est pas le cas, c'est qu'il y a eu une attaque avec modification du message entre l'agent *a* et l'agent *b*.

La Figure 3.9 décrit la fonction trouver_coupable() qui est exécutée par l'agent *b* lorsqu'il détecte une attaque avant l'étape *n*. La Figure 3.10 présente l'algorithme exécuté à l'étape n. Il est exécuté si aucune attaque n'a été détectée lors des étapes précédentes.

```
Initialisation :   a est sur h0
                   i := 1
Tant que           hai ≠ h0 faire
                        Communiquer à hbi( preva_i, nexta_i, sign_hai (preva_i ||
nexta_i) )
                        Si communication_reussie( ) = faux
                             alors migrer ( h0 )
                        Sinon tache ( )
                             i := i +1
                             migrer ( hai )
                        Fin si
Fin tant que
```

Figure 3.7 Algorithme pour l'agent a

```
Initialisation :   b est sur h0
                   i := 0
                   Continuer := vrai
Tant que           Continuer faire
                        Attendre_communication ( )
                        Si verifier_signature ( prevai || nextai , signhai) = faux
                           OU id(hai) ≠ nextai – 1
                           OU prevai ≠ id (hai – 1)
                              alors   migrer ( h0 )
                                      trouver_coupable()
                                      Continuer : = faux
                        Sinon  enregistrer_itinéraire ( prevai,
                                      nextai,
                                        signhai (prevai || nextai),
                                        signhbi (prevai || nextai||signhai) )
                               Si nextai = h0
                                      alors   migrer ( h0 )
                                              Continuer : = faux
                               Sinon  i : = i + 1
                                      migrer ( hbi )
                                Fin si
                        Fin si
Fin tant que
```

Figure 3.8 Algorithme pour l'agent b

Initialisation : coupable1 : = personne

coupable2 : = personne

Si verifier_signature ($preva_i$ || $nexta_i$, $sign_{hai}$) = faux

Alors coupable1 : = ha_i

Sinon **Si** $id(ha_i) \neq nexta_{i-1}$

Alors coupable1 : = ha_{i-1}

Sinon **Si** $preva_i \neq id\ (ha_{i-1})$

Alors **Si** id ($sign_{hai-1}$) = $nexta_{i-2}$

Alors coupable1 : = $preva_i$

coupable2 : = ha_{i-1}

Sinon coupable1 : = ha_{i-2}

coupable2 : = hb_{i-1}

Fin si

Fin si

Fin si

Fin si

Figure 3.9 Fonction trouver_coupable()

coupable1 : = personne
coupable2 : = personne
coupable3 : = personne
validite : = vrai
i : = 1
Tant que coupable1 = personne **ET** i < n +1 **ET** validité = vrai **faire**
Si verifier_signature ($preva_i \| nexta_i$, $sign_{hai}$) = vrai
ET verifier_signature ($preva_i \| nexta_i \| sign_{hai}$, $sign_{hbi}$) = vrai
alors **Si** $id(sign_{hai}) \neq preva_{i+1}$
alors coupable1 : = ha_i
coupable 2 : = hb_{i+1}
Fin si
Si $id(sign_{hai}) \neq nexta_{i-1}$
alors coupable1 : = ha_{i-1}
coupable2 : = hb_i
coupable3 : = hb_{i+1}
Fin si
Si $id(sign_{hai}) = preva_{i+1}$
ET $id(sign_{hai}) = nexta_{i-1}$
ET plate_forme_conforme($id(sign_{hai})$) = faux
alors coupable1 : = ha_{i-1}
Sinon validite : = faux
Fin si
i : = i + 1
Fin Tant que

Figure 3.10 Algorithme étape n

3.2.3 Sécurité du protocole

Le protocole fonctionne de la même manière que celui de Roth (1998) et détecte les mêmes attaques lorsqu'on fait les mêmes hypothèses que ce dernier, c'est-à-dire si on suppose qu'il n'y a pas de collaboration. Cependant, nous allons voir qu'avec les hypothèses plus faibles que nous avons émises, on arrive à des résultats identiques au niveau des attaques détectées.

Pour bien passer en revue toutes les possibilités de collaboration, nous allons décomposer l'ensemble des collaborations possibles en quatre. Les possibilités de collaboration entre deux hôtes ha_i et hb_j sont inclues dans l'un de ces quatre ensembles :

- $i < j - 1$,
- $i = j - 1$,
- $i = j$,
- $i > j$.

Enfin, nous envisagerons le cas d'une collaboration entre ha_i, hb_{i+1} et hb_{i+2}. C'est le seul cas que nous étudions avec une collaboration entre une plate-forme de l'itinéraire de *a* et deux plates-formes de l'itinéraire de *b*. En effet, les autres attaques avec collaboration de ce type peuvent être modélisés comme deux attaques différentes avec collaboration entre une plate-forme de l'itinéraire de l'agent *a* et une de l'itinéraire de l'agent *b*.

Attaque avec collaboration entre ha_i et hb_j, avec $i < j - 1$: attaque avec modification de l'enregistrement de l'itinéraire

Lorsque l'agent *a* est sur l'hôte ha_i et que cet hôte fait une attaque avec modification du message entre *a* et *b*, comme nous l'avons vu précédemment, l'attaque n'est détectée que lors du retour des agents sur la plate-forme de leur propriétaire. Il serait donc intéressant pour l'attaquant ha_i de pouvoir modifier l'enregistrement de l'itinéraire que transporte l'agent *b* de manière à ce que l'attaque ne soit pas détectée. Il peut donc demander à la plate-forme hb_j de modifier l'entrée correspondante à l'attaque en la remplaçant par l'identité où voulait aller l'agent. De cette manière, l'attaque ne sera pas détectée lors du retour de l'agent. La Figure 3.11 illustre cette attaque.

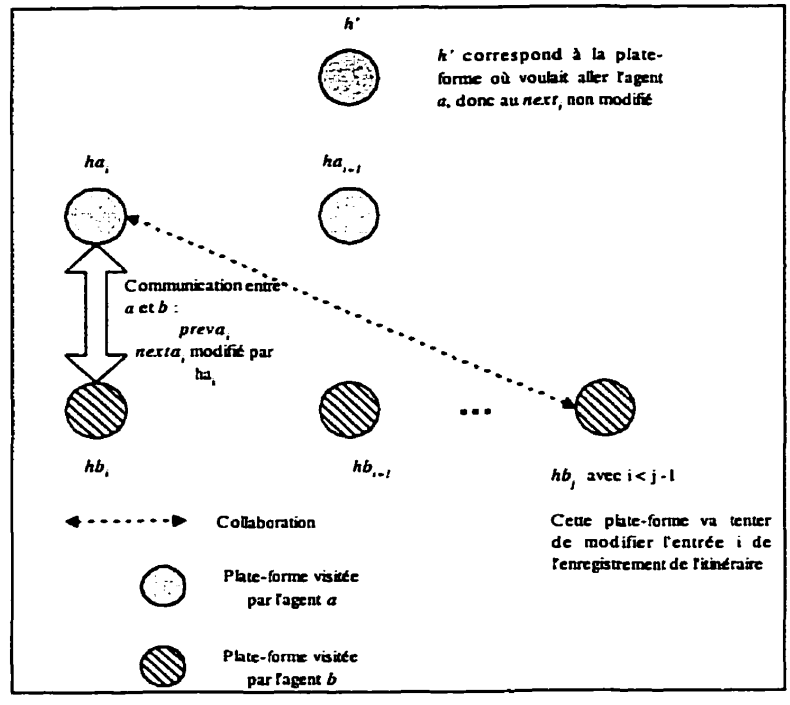

Figure 3.11 Attaque avec collaboration entre ha_i et hb_j avec $i < j - 1$

Cependant, ce remplacement d'une entrée dans l'enregistrement de l'itinéraire est impossible car il est résistant aux attaques de modification. En effet, une entrée dans ce dernier est signée à la fois par la plate-forme ha_i qui a envoyé l'identité de la prochaine plate-forme où voulait aller l'agent *a*, $nexta_i$, et par la plate-forme où est l'agent *b* au moment où il a écrit dans l'itinéraire, hb_i. Si une plate-forme hb_j tente de modifier la $i^{ième}$ entrée avec $j > i$, il ne pourra pas créer une entrée valide au niveau des signatures car il ne peut pas signer à la place de la plate-forme hb_j. Donc, si jamais il modifie $nexta_i$; la signature ne sera plus valide et l'attaque sera détectée lors de la vérification des signatures par la plate-forme du propriétaire des agents. Par contre, il sera impossible de trouver le coupable. Si hb_i collabore avec hb_j pour qu'il puisse signer à sa place, l'attaque sera quant même détectée au retour de l'agent car la correspondance entre $nexta_i$ et l'identité de la plate-forme qui a signé l'entrée i+1 de l'enregistrement de l'itinéraire n'est pas valable.

S'il est important de trouver le coupable de cette attaque, il est possible de faire effectuer la vérification de l'ensemble des entrées de l'itinéraire après chaque migration

de l'agent *b*. Ainsi, si une entrée a été modifiée, cette modification sera détectée et le coupable sera la plate-forme précédente sur l'itinéraire de *b*. Nous n'avons pas choisi cette solution car elle est très coûteuse en temps d'exécution : vérifier une signature est coûteux et il faudrait répéter cette opération pour l'ensemble des entrées de l'enregistrement à chaque migration de l'agent *b*. C'est pourquoi nous avons choisi de faire la vérification uniquement lors du retour de l'agent *b* sur la plate-forme de son propriétaire, même si nous ne pouvons alors désigner le coupable.

Attaque avec collaboration entre ha_i et hb_j, avec $i = j - 1$

Si ha_i envoie l'agent *a* vers une plate-forme ha_{i+1} avec $id(ha_{i+1}) \neq nexta_i$, la plate-forme hb_{i+1} va recevoir les informations $id(ha_{i+1})$, $preva_{i+1}$ et $nexta_{i+1}$, puis va vérifier l'égalité $id(ha_{i+1}) = nexta_i$. Or, à moins qu'il y ait une mascarade par vol de clé d'authentification, ce qu'on suppose impossible, cette égalité n'est pas vérifiée. Donc, l'agent *b* qui est sur la plate-forme hb_{i+1} trouve qu'il y a un problème sur l'itinéraire.

Cependant, il se peut que cette plate-forme collabore avec ha_i pour faire croire à l'agent *b* que cette égalité est vraie. La Figure 3.12 illustre cette attaque.

On a donc ha_i qui collabore avec hb_{i+1}. L'agent *b* fait la vérification des égalités à l'étape i+1, mais il ne détecte pas l'erreur car la plate-forme sur laquelle il s'exécute hb_{i+1} le manipule. L'agent trouve donc que les deux égalités sont vérifiées.

Une fois sur la plate-forme hb_{i+2}, la vérification de l'égalité $preva_{i+2} = id\ (ha_{i+1})$ va donner une erreur car la plate-forme ha_i n'a pas envoyé l'agent vers $nexta_{i+1}$. Pour trouver le coupable, il se pose un problème car cette égalité qui n'est pas vérifiée correspond aussi au cas où l'hôte ha_{i+1} envoie l'agent à une mauvaise plate-forme qui tente de le remettre sur sa route. La Figure 3.13 illustre ce cas. En effet, si ha_{i+1} envoie l'agent *a* vers une plate-forme différente de $nexta_{i+1}$ et que cette plate-forme envoie l'agent *a* vers la plate-forme d'identité $nexta_{i+1}$ pour remettre l'agent sur la route, une fois sur la plate-forme ha_{i+2}, l'égalité $prev_{i+2} = id(ha_{i+1})$ ne sera pas vérifiée, car cette dernière connaît de manière authentifiée l'identité de la plate-forme qui lui envoie l'agent *a*.

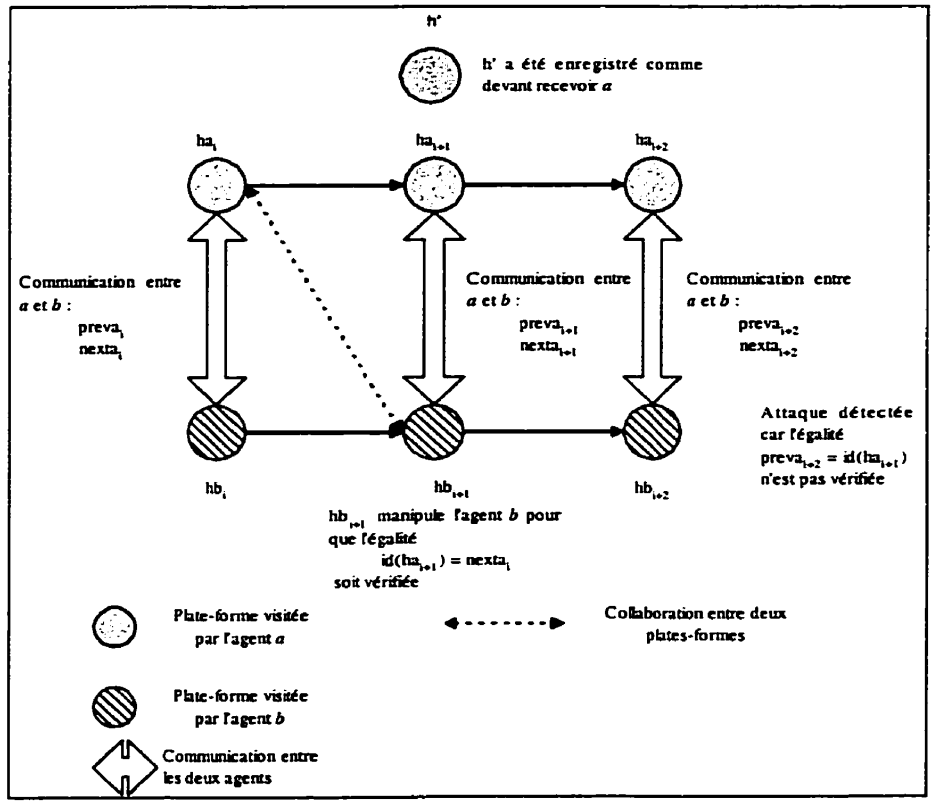

Figure 3.12 Attaque avec collaboration entre ha_i et hb_j avec i = j – 1

Nous avons donc un problème ici pour désigner le coupable : nous savons qu'il est possible que ce soit ha_i, c'est le cas que nous avons étudié avec une collaboration de hb_{i+1} ou qu'il s'agit de ha_{i+1} qui, avec une plate-forme complice, a tenté de remettre l'agent *a* sur sa route. Pour cela, il faut que l'agent *b* vérifie la signature de $nexta_{i+1}$ par ha_{i+1} et compare cette identité avec $nexta_i$. Cela est fait dans la fonction trouver_coupable() qui est exécutée lorsque l'agent détecte l'attaque. Si on est dans le cas de la première attaque, l'identité de la plate-forme qui a signé $nexta_{i+1}$ est différente de $nexta_i$ et le coupable est ha_i, avec une collaboration de hb_{i+1}. Si on est dans le deuxième cas, on aura égalité entre ces deux entités et le coupable est ha_{i+1}. La Figure 3.14 montre les entrées i et i+1 de l'enregistrement de l'itinéraire lorsqu'on est dans le

cas de l'attaque avec collaboration et la Figure 3.15 montre ces mêmes entrées dans le cas de l'attaque sans collaboration.

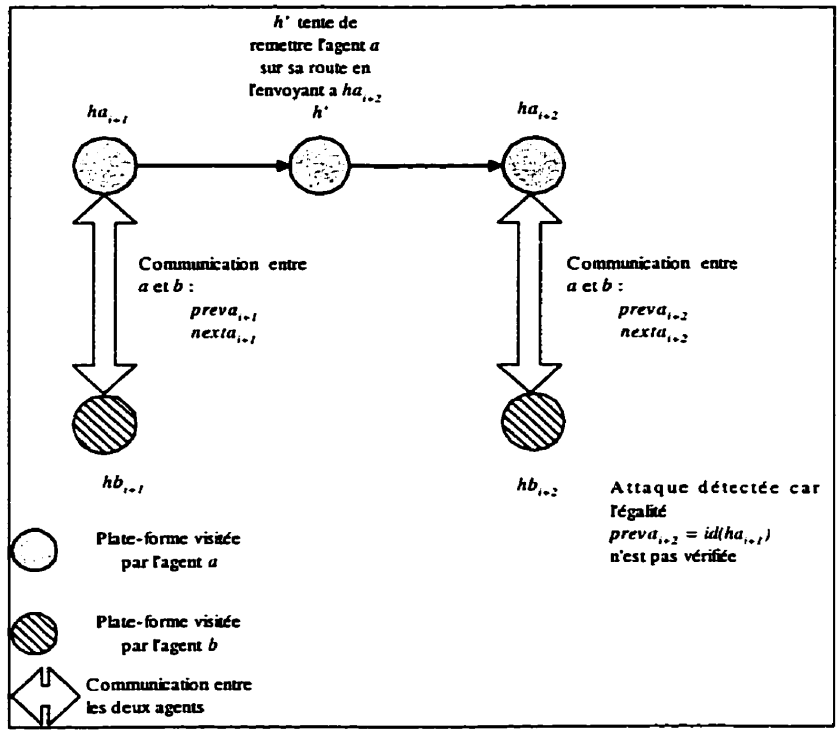

Figure 3.13 Attaque où ha_{i+1} envoie l'agent vers une mauvaise plate-forme

$preva_i$	$nexta_i = id(h')$	$sign_{hai}(nexta_i \| preva_i)$	$sign_{hbi}(preva_i \| nexta_i \| sign_{hai})$
$preva_{i+1}$	$nexta_{i+1}$	$sign_{hai+1}(nexta_{i+1} \| preva_{i+1})$	$sign_{hbi+1}(preva_{i+1} \| nexta_{i+1} \| sign_{hai+1})$

Figure 3.14 Entrées i et i+1 de l'enregistrement de l'itinéraire dans le cas de l'attaque avec collaboration

$preva_i$	$nexta_i = id\ (h_{i+1})$	$sign_{hai}(nexta_i \| preva_i)$	$sign_{hbi}\ (preva_i \| nexta_i \| sign_{hai})$
$preva_{i+1}$	$nexta_{i+1}$	$sign_{hai+1}(nexta_{i+1} \| preva_{i+1})$	$sign_{hbi+1}\ (preva_{i+1} \| nexta_{i+1} \| sign_{hai+1})$

Figure 3.15 Entrées i et i+1 de l'enregistrement de l'itinéraire dans le cas de l'attaque sans collaboration

Une autre attaque est possible avec collaboration ha_i et hb_j avec i = j – 1. L'hôte ha_i envoie l'agent *a* à h' dont l'identité est différente de celle de l'hôte où veut réellement aller *a*. Ensuite, cet hôte h' tente de remettre l'agent sur sa route en l'envoyant à ha_{i+1}. Une fois sur ce dernier, l'agent *a* communique avec *b* pour lui donner $preva_{i+1}$ et $nexta_{i+1}$. L'agent *b* fait ses vérifications et doit normalement déduire que l'agent a été dévié de sa trajectoire car on a $preva_{i+1} \neq id\ (ha_i)$. Mais, hb_{i+1} le manipule de telle sorte que cette égalité soit vérifiée. La Figure 3.16 illustre cette attaque.

Cette attaque n'est pas détectée tout de suite, mais au retour des deux agents sur leur plate-forme d'origine. À l'étape n, la vérification de l'enregistrement de l'itinéraire établira que $id(sign_{hai}) \neq preva_{i+1}$ et les coupables ha_i et hb_{i+1} sont trouvés. Si hb_{i+1} modifie l'enregistrement de l'entrée $i+1$, il ne pourra reproduire la signature de ha_{i+1}. Donc, hb_{i+1} ne peut le faire sans détection au retour de l'agent car, à l'étape n, on vérifie la validité des signatures. Cependant, dans ce cas, le coupable n'est pas trouvé.

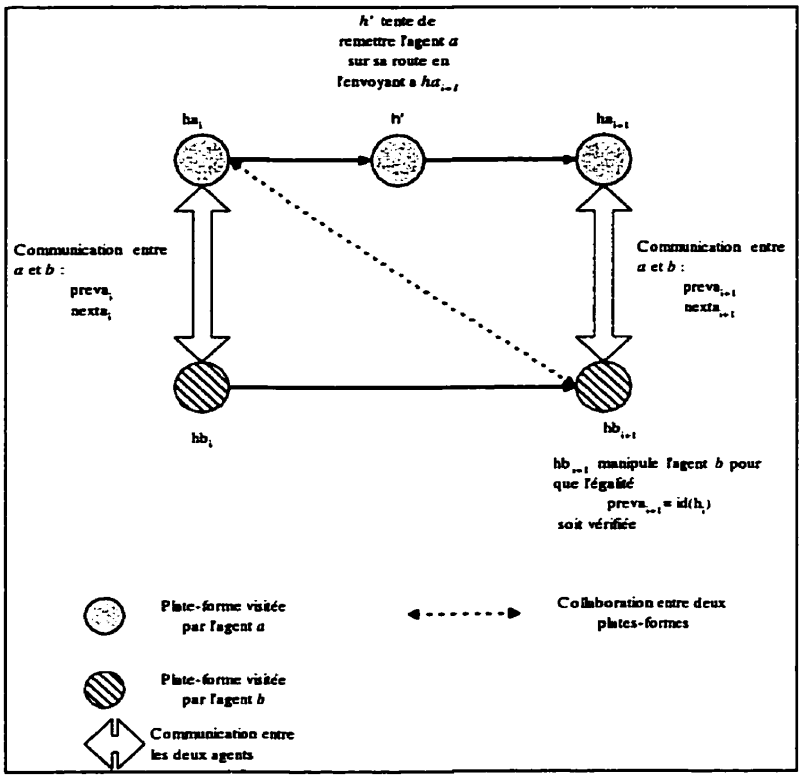

Figure 3.16 Attaque avec tentative de remettre l'agent sur sa route

Attaque avec collaboration entre ha_i et hb_j, pour $j = i$

Si ha_i collabore avec hb_i pour que l'agent b enregistre comme identité de la $i+1^{ème}$ plate-forme visitée par l'agent a, $nexta_i$, non pas la plate-forme où veut aller a qu'on appelle h', mais là où la plate-forme attaquante ha_i veut l'envoyer. La Figure 3.17 illustre cette attaque. L'agent b enregistre alors un $nexta_i$ qui est faux et la plate-forme ha_i envoie l'agent sur la plate-forme qui a été enregistrée. Dans ce cas, lorsque b va être sur la plate-forme hb_{i+1}, qu'on suppose honnête vis à vis du protocole, il va recevoir les informations suivantes de la part de a : $nexta_{i+1}$, $preva_{i+1}$, $id(ha_{i+1})$. La vérification de l'égalité $nexta_i = id\ (ha_{i+1})$ est effectuée et ne permet pas de détecter l'attaque pour le moment car la plate-forme hb_i a enregistré, non pas où l'agent voulait aller (h'), mais où la plate-forme ha_i l'a envoyé (ha_{i+1}). Cependant, cette fraude sera détectée lors du retour de l'agent sur sa plate-forme d'origine car l'agent b a conservé l'itinéraire complet de a et le propriétaire de l'agent vérifie que l'itinéraire de celui-ci est conforme à sa fonction.

Cela n'est pas un handicap pour notre protocole car cette fraude est équivalente à celle où une plate-forme malhonnête falsifie $nexta_i$, dans la communication entre *a* et *b* pour envoyer l'agent *a* vers la plate-forme qu'elle désire, sans détection de la part de *b*. Nous avons nommé cette attaque « attaque avec modification de la communication entre *a* et *b* » dans le protocole de Roth. Ce dernier ne permet pas non plus de détecter immédiatement cette attaque, elle la détecte à l'étape n en vérifiant l'itinéraire de l'agent *a* au retour de l'agent *b*. Si ha_{i+1} n'est pas honnête vis à vis du protocole, la plate-forme doit tenter de remettre l'agent sur sa route. Cela est alors détecté par hb_{i+1} car l'égalité $preva_{i+2} = id(ha_{i+1})$ ne sera pas vérifiée.

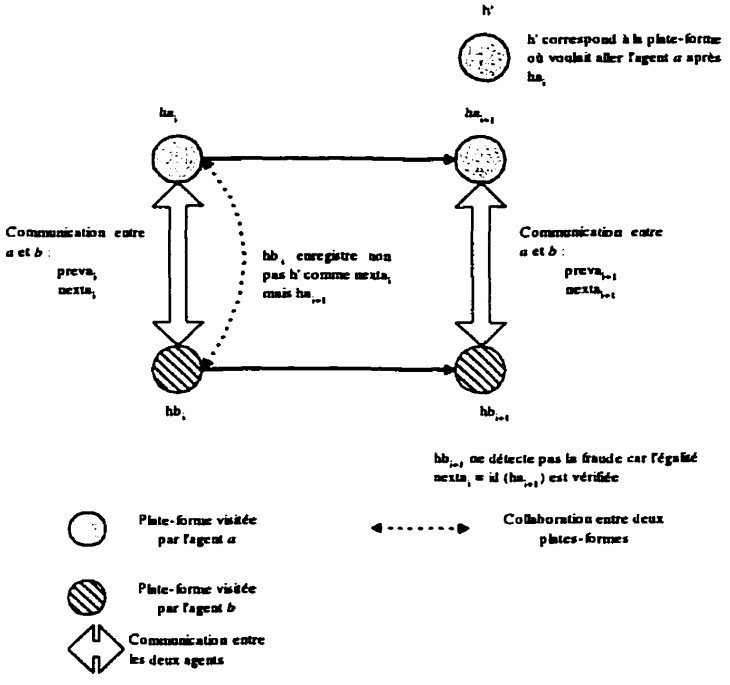

Figure 3.17 Attaque avec collaboration entre ha_i et hb_j pour j = i

Attaque avec collaboration entre ha_i et hb_j, pour $i > j$

Une collaboration de ce type ne permet pas de faire une attaque. En effet, l'agent *b* est exécuté sur l'hôte hb_j avant que l'agent *a* n'ait migré sur l'hôte ha_i. La Figure 3.18 illustre ces hypothèses.

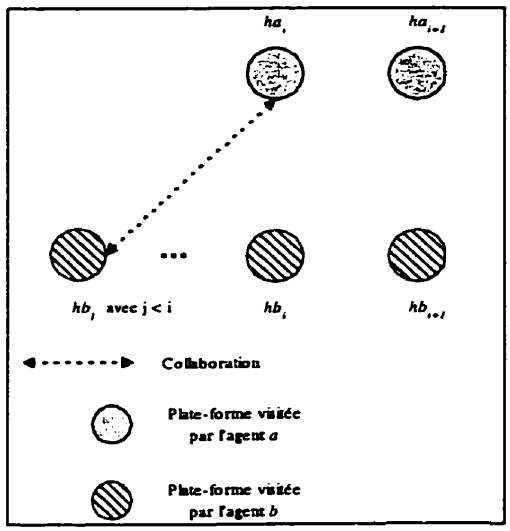

Figure 3.18 Attaque impossible pour la collaboration entre ha_i et hb_j, avec i > j

Attaque avec collaboration entre ha_i, hb_{i+1} et hb_{i+2}

Le principe de cette attaque est la même que celle avec collaboration entre ha_i et hb_{i+1} : ha_i envoie l'agent *a* vers une plate-forme différente de celle où il veut aller; une fois sur cette plate-forme, l'agent *a* communique avec l'agent *b* pour lui donner les identités des plates-formes, comme il est prévu dans le protocole. Normalement, *b* doit détecter l'attaque mais il est manipulé par la plate-forme hb_{i+1} qui collabore avec ha_i. Il ne détecte donc pas l'attaque et migre vers l'hôte hb_{i+2}. Dans le cas où hb_{i+2} était honnête, nous aurions détecté l'attaque immédiatement et nous aurions trouvé les coupables. Ici, hb_{i+2} collabore et manipule l'agent *b* pour qu'il croie que l'égalité $preva_{i+2} = id(ha_{i+1})$ est vérifiée. La Figure 3.19 illustre cette attaque. Dans ce cas, pour détecter l'attaque, il faut attendre le retour des agents sur la plate-forme de leur propriétaire. En effet, l'algorithme exécuté lors du retour de l'agent sur la plate-forme du propriétaire (étape n) compare l'identité de la plate-forme sur l'itinéraire de *a* qui a signé l'entrée i+1 avec l'identité de $nexta_i$, la plate-forme où voulait aller l'agent. La Figure 3.20 montre les entrées i et i+1 de l'enregistrement de l'itinéraire; on voit que $nexta_i$ est différent de l'identité de la plate-forme qui a signé l'entrée i+1 de l'itinéraire (ha_{i+1}). Si cette attaque n'a pas été détectée auparavant, c'est qu'il y a eu collaboration entre ha_i,

hb_{i+1} et hb_{i+2}. En effet, comme nous l'avons vu précédemment, si hb_{i+2} est honnête, cette plate-forme détectera l'attaque et désignera les coupables.

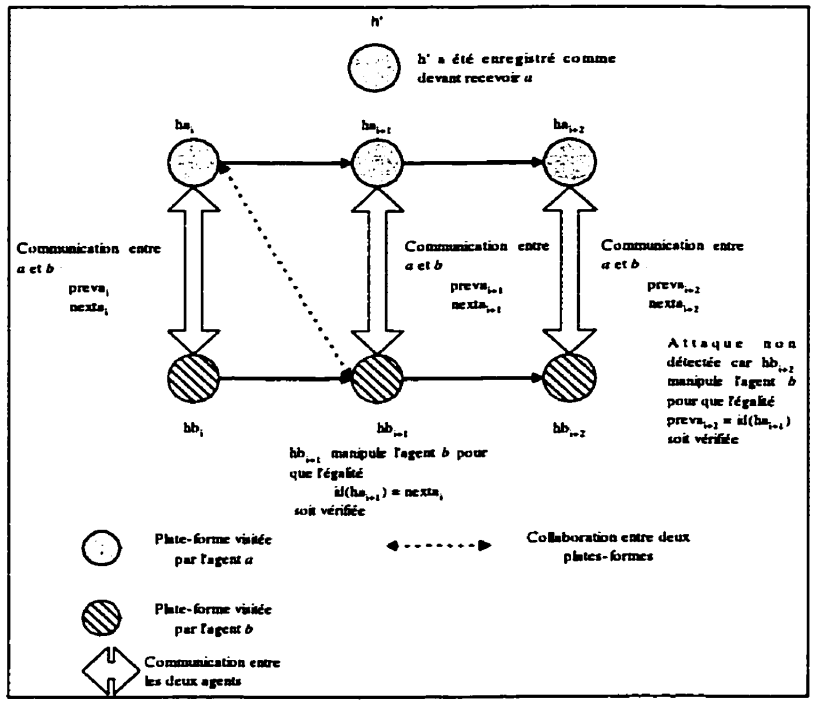

Figure 3.19 Attaque avec collaboration entre ha_i, hb_{i+1} et hb_{i+2}

$preva_i$	$nexta_i = id(h')$	$sign_{hai}(nexta_i \| preva_i)$	$sign_{hbi}(preva_i \| nexta_i \| sign_{hai})$
$preva_{i+1}$	$nexta_{i+1}$	$sign_{hai+1}(nexta_{i+1} \| preva_{i+1})$	$sign_{hbi+1}(preva_{i+1} \| nexta_{i+1} \| sign_{hai+1})$

Figure 3.20 Entrées i et i+1 de l'enregistrement de l'itinéraire dans le cas d'une attaque avec collaboration entre ha_i, hb_{i+1} et hb_{i+2}

Justification de nos choix

Le choix de faire signer, à chaque étape, l'entrée de l'itinéraire par la plate-forme qui a exécuté l'agent b nous permet de savoir de manière certaine sur quelle plate-forme

l'agent b s'est exécuté. Comme son itinéraire est fixé dès le départ, il est facile de vérifier lors du retour de l'agent l'identité de chacun des hôtes qui l'a exécuté. De plus, cette signature permet d'empêcher la falsification d'entrées de l'itinéraire effectuées par des plates-formes se suivant sur cet itinéraire. Si cela n'était pas le cas, un hôte pourrait faire accuser des plates-formes qui sont innocentes. En effet, avec la signature, on est sûr qu'une entrée valide dans l'enregistrement de l'itinéraire a été effectuée par l'hôte qui prétend l'avoir fait. En d'autres termes, il n'est pas possible d'usurper l'identité de la plate-forme qui a signé l'entrée de l'itinéraire. De la même manière, la signature de la plate-forme qui a exécuté l'agent *a* nous permet d'être certain de cette identité et d'empêcher les modifications. Ainsi, il est impossible de faire croire qu'une plate-forme ha_i a exécuté l'agent si ce n'est pas le cas car l'entrée de l'itinéraire correspondante ne sera pas signée par cette plate-forme et lors du retour des agents sur la plate-forme de leur propriétaire, il y a vérification de la correspondance entre les identités enregistrées et les signatures.

Le choix de faire se déplacer l'agent *b* à chaque fois que l'agent *a* migre se justifie pour deux raisons, la désignation des coupables et le moment de la détection, sachant que ces deux raisons sont liées. Le principal autre cas possible sur les déplacements de l'agent *b* est que ce dernier reste sur le même hôte pendant que l'agent a effectue sa tâche, cet hôte, qu'on appelle *hb* par la suite, est connu dès le départ des deux agents et l'agent *a* communique toujours avec l'agent *b* sur cette plate-forme. Il est évident que, pour n'importe quelle attaque à laquelle *hb* prend part, il faudra attendre le retour de l'agent *b* sur la plate-forme du propriétaire des agents pour pouvoir détecter l'attaque. Or, avec nos déplacements, la majorité des attaques effectuées à l'étape i sont détectées à l'étape i +1 ou i+2.

De plus, avec un seul hôte où s'exécute l'agent *b*, quelle que soit l'attaque, le protocole ne sera pas capable de donner l'identité de l'hôte de l'itinéraire de l'agent *a* avec qui *hb* a collaboré. En effet, au retour de l'agent sur la plate-forme de son propriétaire, si l'enregistrement de l'itinéraire comporte des incohérences (par exemple au niveau des signatures), c'est que la plate-forme où s'est exécuté l'agent *b* a effectué

une attaque, car sinon elle aurait dû détecter cette incohérence et l'agent aurait migré directement sur la plate-forme de son propriétaire. Maintenant qu'on sait cela, il est impossible de donner de manière certaine l'identité de la ou des plates-formes sur l'itinéraire de l'agent *a* qui ont collaboré pour modifier l'itinéraire de l'agent. En effet, on ne sait pas exactement à quelle étape l'attaque a eu lieu et on ne peut le déduire à partir des incohérences de l'enregistrement de l'itinéraire car ce dernier a pu être modifié par la plate-forme *hb* dans le but de rendre impossible la désignation des coupables ou pour faire accuser une plate-forme innocente qui a exécuté l'agent *a*.

Si notre protocole détecte une attaque et peut désigner les coupables, c'est dans le cas où l'hôte suivant la plate-forme malveillante sur l'itinéraire de l'agent *b* est honnête et fait les vérifications demandées par le protocole. Du fait qu'on sait à quelle étape a eu lieu l'attaque, on peut plus facilement déduire l'identité des coupables. Dans le cas où cet hôte n'est pas honnête et collabore avec la précédente pour ne pas déceler l'attaque, cette dernière sera détectée au retour de l'agent et les coupables seront désignés dans certains cas. On peut supposer qu'il y a peu de chances pour que deux plates-formes consécutives sur l'itinéraire de l'agent *b* soient malveillantes et collaborent. En effet, cet itinéraire est fixé dès le départ des agents et on a choisi les plates-formes visitées par l'agent *b* et leur ordre de manière aléatoire parmi un ensemble de plates-formes qu'on sait que l'agent *a* ne visitera pas.

3.2.4 Apport de notre protocole par rapport à celui de Roth

Résolution du problème d'échange d'agent

Notre protocole permet d'éviter un problème mis en avant par Roth concernant son protocole : si une plate-forme ha_{i+1} reçoit deux agents a_1 et a_2 venant de la même plate-forme ha_i (cf. Figure 3.21), la plate-forme pouvait intervertir les agents « enregistreurs » d'itinéraire b_1 et b_2 de manière à ce que a_1 communique $preva_{1\ i+1}$, $nexta_{1\ i+1}$ et $id(ha_{i+1})$ à b_2 et a_2 communique $preva_{2i+1}$, $nexta_{2i+1}$ et $id(ha_{i+1})$ à b_1. Chacun des agents enregistreurs b_1 et b_2 va faire les vérifications des deux égalités $id(ha_{i+1}) = next_i$ et $prev_{i+1} = id\ (ha_i)$ respectivement pour l'agent *a2* et *a1*, il n'y aura pas de

problème car les deux agents a_1 et a_2 viennent de la même plate-forme ha_i. Une fois sur la plate-forme suivante i+2, chacun des deux agents a_1 et a_2 vont communiquer à leurs agents coopérants respectifs les identités next et prev. À partir de ce moment, toutes les vérifications des agents enregistreurs b_1 et b_2 seront valides si le protocole est respecté. On ne pourra détecter l'inversion des agents que lors du retour des agents sur la plate-forme d'origine. Roth évoque une solution à ce problème sans la détailler : il faut que les agents s'identifient mutuellement. Cependant, cette solution demande à être rajoutée au dessus du protocole, ce qui n'est pas le cas avec notre protocole. En effet, l'agent a connaît l'itinéraire de son agent coopérant b, cet itinéraire est protégé contre les altérations et est lié à l'agent. Il n'est donc pas possible d'interchanger les agents « enregistreurs » d'itinéraire de l'agent a_1 et a_2.

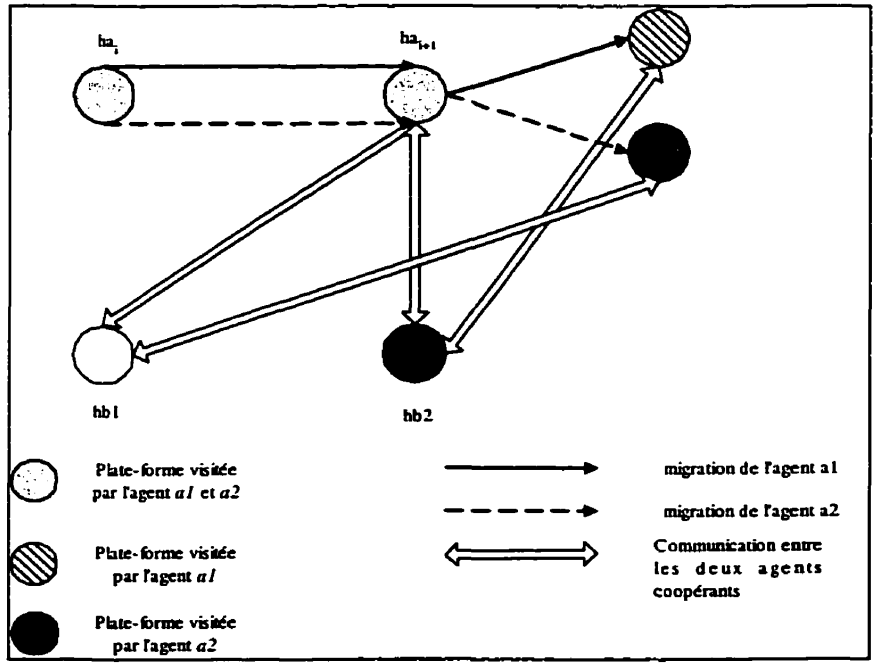

Figure 3.21 Problème du changement d'agent enregistreur avec le protocole original

Quantification de l'amélioration du pouvoir de détection des attaques

Soit p_{ij} la probabilité pour que l'hôte i sur l'itinéraire de l'agent *a* (appelé ha_i) et l'hôte j sur l'itinéraire de l'agent *b* (appelé hb_j) collabore. Soit P_c la probabilité pour qu'il y ait au moins une collaboration entre un hôte sur l'itinéraire de l'agent *a* et un hôte sur l'itinéraire de l'agent *b*. Désignons par n le nombre de déplacement de l'agent.

$$P_c = 1 - \prod_{j=0}^{n} \prod_{i=0}^{n} (1 - p_{ij})$$

Or, la plate-forme du propriétaire des agents est, par hypothèse, supposée de confiance; elle correspond à la plate-forme visitée à l'étape 0 et à l'étape n par les deux agents coopérants. On a donc :

$\forall j \in \{0, \ldots, n\}, p_{0j} = 0$

$\forall j \in \{0, \ldots, n\}, p_{nj} = 0$

$\forall i \in \{0, \ldots, n\}, p_{i0} = 0$

$\forall i \in \{0, \ldots, n\}, p_{in} = 0$

On obtient :

$$P_c = 1 - \prod_{j=1}^{n-1} \prod_{i=1}^{n-1} (1 - p_{ij})$$

On en déduit la probabilité P_R pour qu'il n'y ait pas de collaboration entre un hôte sur l'itinéraire de l'agent *a* et un hôte sur l'itinéraire de l'agent *b*. Cette probabilité correspond à la probabilité pour que les hypothèses de Roth soient vérifiées :

$$P_R = 1 - P_c = \prod_{j=1}^{n-1} \prod_{i=1}^{n-1} (1 - p_{ij})$$

Si on suppose que la probabilité est identique quel que soit le choix du couple (i,j), ce qui signifie : $\forall (i,j) \in \{1, \ldots, n-1\}, p_{ij} = p$

On obtient :

$$P_c = 1 - (1 - p)^{(n-1)^2}$$

$$P_R = (1 - p)^{(n-1)^2}$$

Si $(n - 1)^2 p << 1$, le développement limité au premier ordre donne :

$$P_R \approx 1 - (n - 1)^2 p$$

Dans ce cas, la probabilité pour qu'une attaque ne soit pas détectée par le protocole de Roth est :

$$P_c \approx (n - 1)^2 p$$

Comme notre protocole détecte les attaques avec collaboration, sa protection est totale. La différence de protection entre notre méthode et celle de Roth est égale à P_c. Il est difficile d'avoir une idée de la valeur de la probabilité p car, à notre connaissance, aucune application utilisant les agents mobiles n'a été déployée sur un réseau public. On ne possède donc pas de statistiques sur les probabilités de collaboration. Cependant, on peut supposer qu'elle est faible car, dans notre protocole, les plates-formes que visite l'agent *b* sont choisies de manière aléatoire parmi les plates-formes où on sait que l'agent *a* ne migrera pas.

CHAPITRE IV
IMPLÉMENTATION, MISE EN ŒUVRE ET ÉVALUATION

Nous avons tenté de faire l'implémentation et la mise en oeuvre de notre protocole de vérification et d'enregistrement d'itinéraire de manière aussi proche que possible d'une implémentation réelle. Nous avons dû choisir un contexte de mise en œuvre, soit principalement une implémentation de plate-forme, car il n'y en a pas encore qui soit devenue un standard pour tous les utilisateurs des agents mobiles. Cela s'explique par le fait que cette technologie est encore très jeune et qu'elle est, pour l'instant, très peu utilisée en dehors des milieux universitaires et de recherche. Comme toute plate-forme, celle que nous avons choisie propose des services : nous en avons utilisé certains mais rejeté d'autres pour utiliser nos propres implémentations.

Dans ce chapitre, nous allons présenter le contexte de la mise en oeuvre du protocole, puis la manière dont nous avons implémenté le protocole. Enfin, nous allons décrire les tests que nous avons effectués pour évaluer cette implémentation, les mesures que nous avons prises et que nous avons analysées pour en déduire l'efficacité de notre protocole, notamment par rapport à celui de Roth.

4.1 Contexte de la mise en oeuvre du protocole

La plate-forme Grasshopper

Grasshopper est la plate-forme d'agents mobiles que nous avons utilisée pour faire l'implémentation de notre protocole d'enregistrement d'itinéraire. Nous avons opté pour cette plate-forme car elle est utilisée par la firme Ericsson qui est associée à ce

travail de recherche. Elle a été développée par la société allemande IKV, la première version a été disponible en août 1998. Il est à noter que l'utilisation de Grasshopper est gratuite pour des fins de recherche. Le langage de développement de la plate-forme est le Java, particulièrement en raison de sa portabilité. La programmation des agents mobiles se fait donc aussi en Java. Grasshopper est conforme au standard de l'OMG (Object Management Group) sur les agents mobiles, i.e. le standard MASIF (Mobile Agent System Interoperability Facility). Ce dernier a été conçu pour assurer l'interopérabilité entre les différentes plates-formes d'agents mobiles.

Grasshopper est un environnement d'agents répartis (Distributed Agent Environment ou DAE). Il est composé de régions, de places, d'agences et de deux types d'agents. La Figure 4.1 montre ces différentes entités et leurs relations. Au cours de l'exposé de notre implémentation, nous détaillerons les services, principalement ceux proposés par l'entité place, que nous avons utilisés ou ceux que nous aurions pu utiliser mais que nous avons rejetés pour des raisons d'opportunité.

L'application d'agent de magasinage

Nous avons choisi d'implémenter une application où l'enregistrement et la vérification d'itinéraire peuvent aider à rendre cette application plus sécuritaire. Il nous paraissait important de vérifier l'application de notre protocole d'enregistrement d'itinéraire dans un contexte proche d'une application réelle. Notre agent fait le magasinage d'un certain nombre de produits à la place de l'utilisateur. Il possède la description des articles qui intéressent l'utilisateur et il se déplace sur les plates-formes de plusieurs vendeurs pour connaître leur prix de vente. La liste du type d'article est supposée connue par les vendeurs, cela pour simplifier le problème. Notre implémentation comprend deux types de produit : le disque compact et le livre, ces deux objets héritent de la classe *BuyObject*, comme le montre la Figure 4.2. Si on souhaite rajouter des types d'objets, il faut les faire hériter de la classe *BuyObject*. Le fonctionnement de l'agent ne sera pas modifié pour autant.

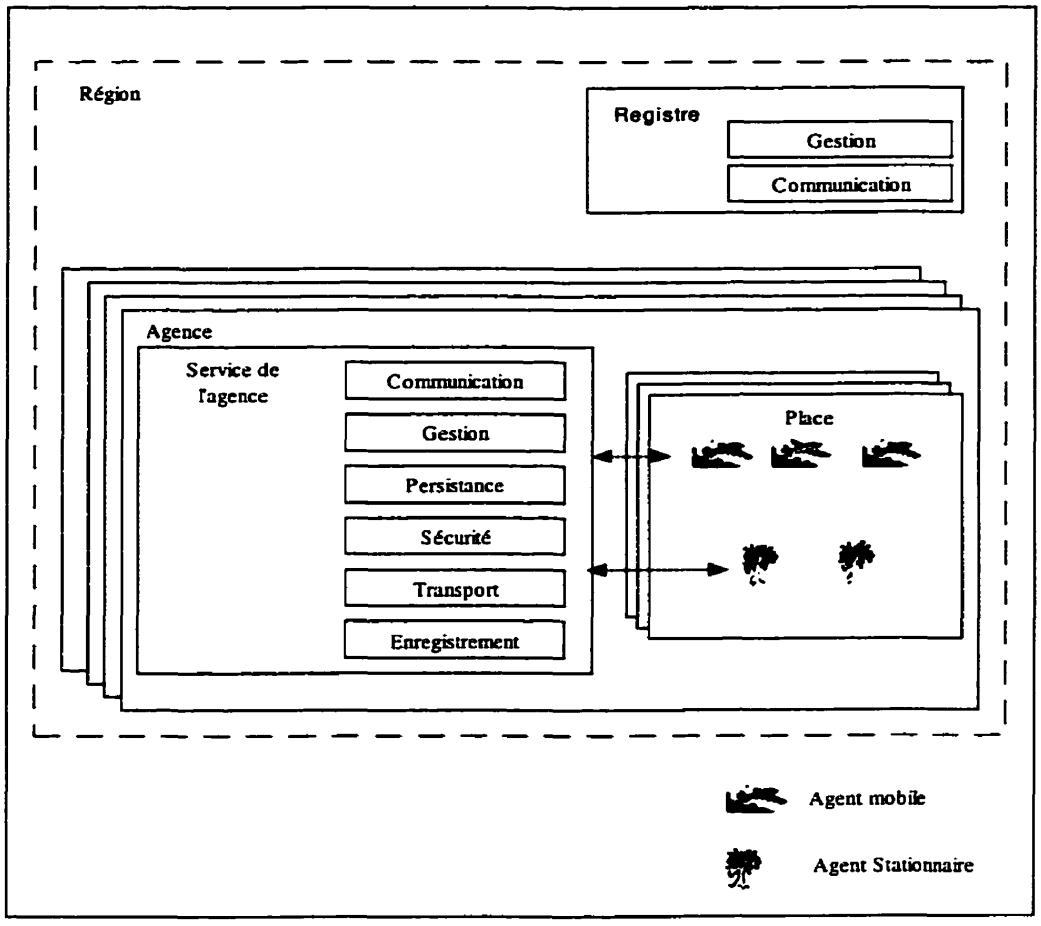

Figure 4.1 Architecture de Grasshopper

L'itinéraire est dynamique : une partie de l'itinéraire est fixée par l'utilisateur au départ, en entrant un certain nombre d'adresses de vendeurs. Mais, pendant la tâche de magasinage, si un des vendeurs n'a pas un produit parce qu'il ne l'a plus en stock ou parce qu'il ne vend pas ce type de produit, il peut proposer à l'agent l'adresse d'un autre vendeur avec qui il a des accords commerciaux. Ces accords peuvent contenir des closes de compensation financière pour le vendeur qui a référé celui qui a vendu le produit. Si cette adresse n'est pas dans l'itinéraire de l'agent qui est composé de l'itinéraire initial et de l'itinéraire des vendeurs ajoutés, il insère cette adresse dans l'itinéraire des vendeurs ajoutés.

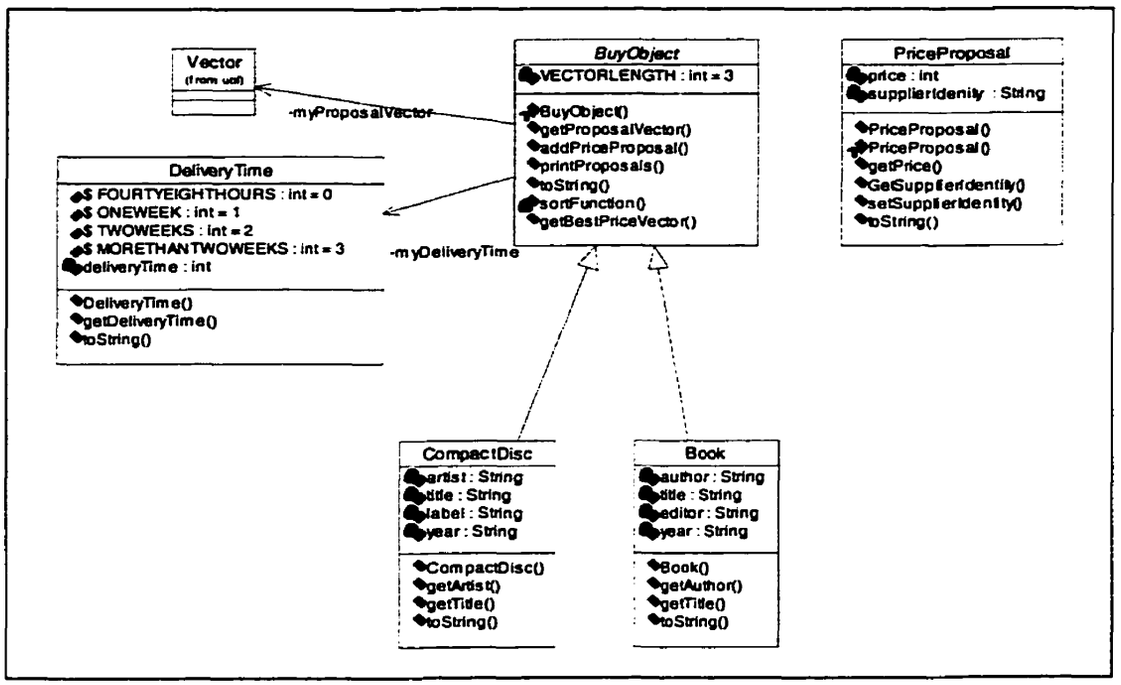

Figure 4.2 Diagramme de classe des produits

La création et l'initialisation de l'agent de magasinage sont faites par un agent stationnaire sur la plate-forme de l'utilisateur. Cet agent ouvre une fenêtre qui est séparée en deux : la partie droite contient la description de la liste des produits que va chercher l'agent, un bouton de type *ComboBox* permet de choisir le type de produit que l'utilisateur veut ajouter, puis une fenêtre s'ouvre invitant l'utilisateur à remplir les différents attributs du type de produit choisi. La partie gauche contient l'itinéraire de l'agent i.e. la liste des adresses des plates-formes qu'il doit visiter. Un bouton permet d'ajouter l'adresse d'une plate-forme à cette liste.

L'agent mobile conserve les trois meilleures offres par produit recherché. Il pourrait conserver l'ensemble des offres mais, si le nombre de vendeurs visités est important, la taille des résultats peut devenir importante. Dans ce cas, il se pourrait que l'agent perde certains de ses avantages par rapport à l'approche client-serveur. Nous avons choisi de garder les trois résultats plutôt qu'un seul car l'utilisateur peut vouloir choisir une compagnie avec un prix un peu plus chère pour son image de marque ou sa réputation. La Figure 4.3 montre un exemple où l'utilisateur veut connaître le prix de

deux produits et où l'itinéraire du départ est fixé à trois vendeurs, dont deux ajoutent l'adresse d'un autre vendeur à l'itinéraire de l'agent.

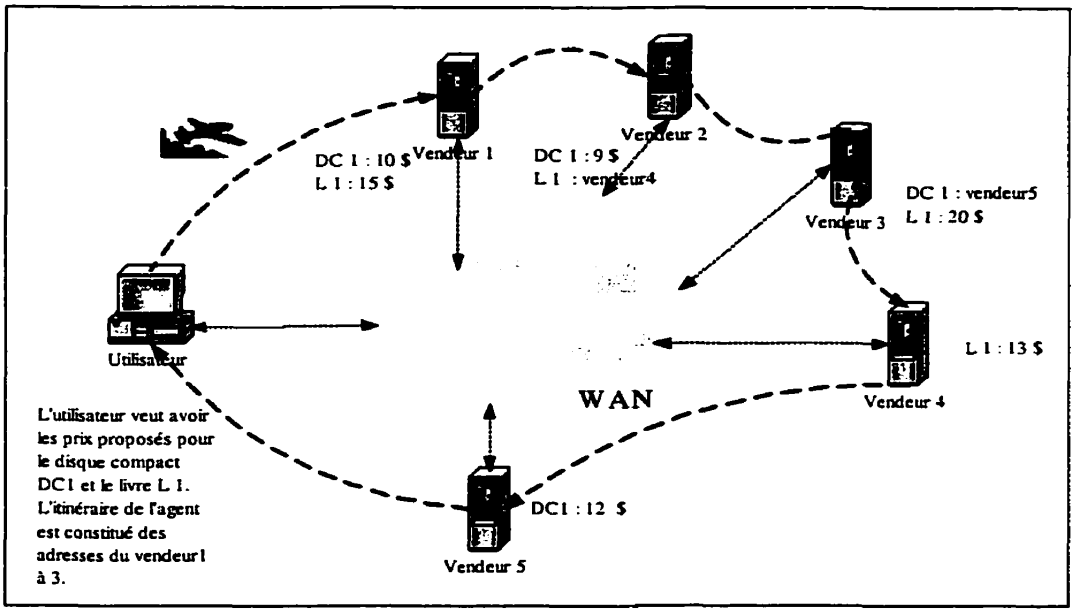

Figure 4.3 Exemple de l'application de magasinage

En ce qui concerne l'implémentation de l'obtention du prix proposé par un vendeur ou l'obtention d'une adresse à rajouter dans l'itinéraire, la meilleure solution aurait été d'utiliser une base de données de type relationnel que l'agent aurait interrogé via JDBC (Java DataBase Connectivity). Cette solution a d'ailleurs déjà été implémentée sous Grasshopper. Cependant, elle demandait un investissement en temps trop important, sachant que la conception de l'agent de magasinage n'était pas le travail principal de notre implémentation. Nous avons donc choisi d'ajouter deux services dans la place où est envoyé l'agent de magasinage : l'un pour demander une proposition de prix (la méthode requestPrice(BuyObject o)), l'autre pour demander une adresse d'un autre vendeur si un produit n'est pas disponible (requestSupplier(BuyObject o)). Pour invoquer les méthodes fournies par une place, l'agent doit être sur cette place (sinon il n'est pas possible d'invoquer ces méthodes) et obtenir son interface par la méthode getInterface() de la classe Place. Par cet intermédiaire, il peut appeler les méthodes qui

sont propres à la place. La base de données associant d'une part un prix et un produit, d'autre part un vendeur et un produit est stockée dans une table de hachage enregistrée sur le disque dur de chaque plate-forme de vendeur. La Figure 4.4 montre le diagramme de classe de ces services.

Maintenant que nous avons vu le contexte de l'implémentation de notre protocole, nous allons voir plus en détails, dans la prochaine section, les problèmes que nous avons rencontrés et, dans la mesure du possible, les choix que nous avons faits pour les résoudre.

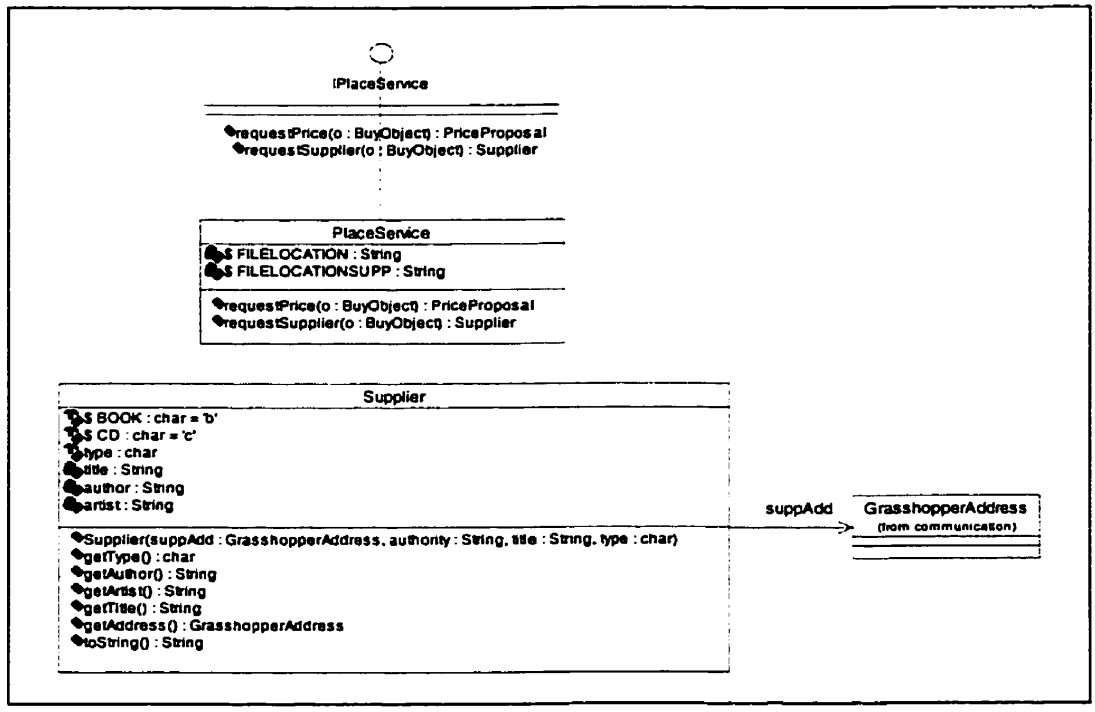

Figure 4.4 Diagramme de classe du service fourni par la plate-forme à l'agent de magasinage

4.2 Mise en oeuvre des hypothèses et choix d'implémentation

H1 : Le transport des agents d'une plate-forme à l'autre se fait par des canaux authentifiés.

Le service de transport de Grasshopper permet d'effectuer le transport des agents via SSL (Secure Socket Layer). Le protocole SSL est maintenant devenu un standard pour les communications où il y a un besoin de sécurité sur Internet. Il utilise l'encryption asymétrique et l'encryption symétrique pour assurer la *confidentialité*, l'*intégrité* et l'*authentification*. La communication entre le client et le serveur est chiffrée avec une clé symétrique et un algorithme qui a été négocié lors de l'ouverture de session (handshake). Cela permet d'assurer la confidentialité. L'intégrité est garantie par l'utilisation de code d'authentification de messages qui prouve qu'un message n'a pas été modifié durant le transport. Le but de l'authentification est d'assurer que chaque paire d'entités communicantes soient mutuellement convaincues de l'identité l'une de l'autre. Pendant l'ouverture d'une session SSL, le client et le serveur échangent leurs identités et leurs clés publiques par l'intermédiaire d'un certificat qui suit la norme X.509 v3. Les identités sont acceptées lorsque le certificat a été signé par une autorité de certification de confiance.

Grasshopper utilise l'algorithme RC4 (Rivest, secret industriel de RSA Data Security, inc., 1987) avec des clés de 128 bits pour l'encryption, l'algorithme RSA (Rivest *et al*, 1983) avec des clés de 1024 bits pour la session d'échange de clés. Ces tailles de clés sont considérées comme sécuritaires par la communauté au moment où est écrit ce mémoire. L'algorithme de code d'authentification des messages utilisé est MD5 (Rivest, 1992). L'utilisation par défaut de SSL par Grasshopper ne force pas l'utilisation de l'authentification, aussi bien au niveau du client qu'au niveau du serveur, cela signifie qu'un client ou un serveur sans certificat peut envoyer ou recevoir l'agent. Étant donné que nous avons besoin d'une authentification mutuelle, nous forçons l'authentification du client et du serveur. Cela signifie que, lors de l'échange des certificats X.509, chacune des entités va vérifier que ce certificat est inclus dans un ensemble de certificats

marqués comme étant de confiance avant d'accepter de continuer d'établir la session SSL.

Le problème de cette approche est qu'il est possible d'accomplir une attaque sur l'utilisation du protocole SSL que fait Grasshopper. Cette attaque est illustrée par la Figure 4.5. Le cas exposé est lorsque c'est un serveur qui fait l'attaque, i.e. on n'envoie pas l'agent à la plate-forme auquel on voulait l'envoyer, le cas où c'est le client qui réalise l'attaque fait croire au serveur que c'est une autre plate-forme qui lui a envoyé l'agent. Soient deux plates-formes susceptibles de recevoir un agent de magasinage, appelées vendeur1 et vendeur2. Ces deux identités sont de confiance pour le propriétaire de l'agent, leurs deux certificats sont marqués comme étant de confiance. Depuis la plate-forme de son propriétaire, l'agent doit migrer vers la plate-forme du vendeur1, puis vers celle du vendeur2. L'agent est envoyé via SSL au vendeur1, cependant le vendeur2 a le contrôle d'un nœud où transitent les paquets et les redirige vers son adresse IP. L'établissement de la session SSL est effectué et le certificat du vendeur2 est reçu par le client qui vérifie que ce certificat est de confiance, ce qui est le cas. L'agent est donc envoyé à la plate-forme du vendeur2 sans détection de l'attaque. Nous avons discuté de ce point avec certains développeurs de Grasshopper par l'intermédiaire d'un forum de discussion. Ils étaient d'accord sur le fait que cette attaque était possible, mais nous ne savons pas s'ils prévoient de modifier leur plate-forme pour empêcher cette attaque dans une prochaine version de Grasshopper. En effet, on peut avancer que ce genre d'attaque est difficile à effectuer et qu'il faut que ce soit une entité dont le certificat fait partie des certificats de confiance qui l'effectue.

On peut voir deux manières d'empêcher cette attaque. La première consiste à ouvrir une fenêtre en donnant l'identité du certificat du serveur avec lequel on a initié la session SSL et en demandant au propriétaire de l'hôte si cette identité correspond à celle de la plate-forme où il veut envoyer l'agent. Le problème de cette technique est qu'elle demande à un utilisateur humain d'intervenir. Or, l'une des caractéristiques des agents mobiles est d'être autonome. Cette technique n'est donc certainement pas la meilleure. La deuxième solution consiste à associer, à chaque migration, un ensemble d'égalités

que devront vérifier les attributs du certificat X.509 (comme le nom, l'entreprise, l'emplacement, le pays ...). La Figure 4.6 montre un diagramme de classe possible pour cette solution : à un vendeur est associée la classe *Supplier*, qui contient l'adresse du vendeur et un objet de type *CertCharacteristic* qui possède un certain nombre d'attributs que le certificat devra vérifier.

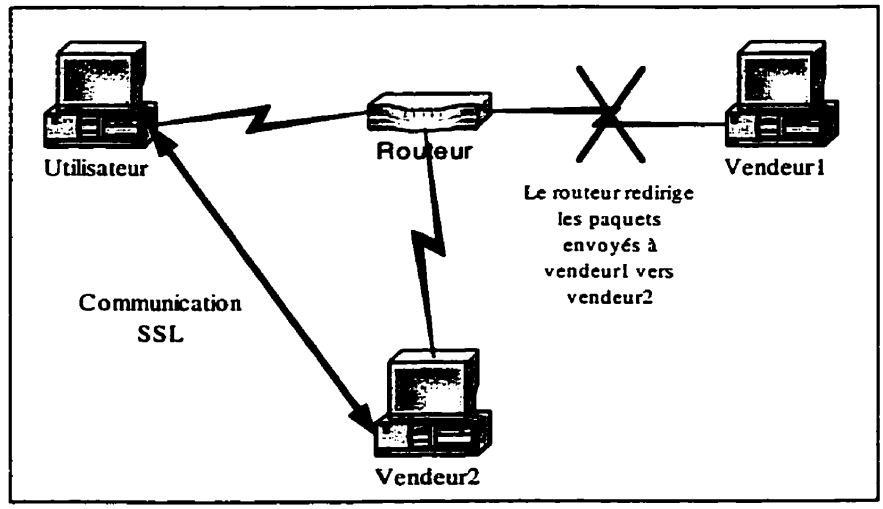

Figure 4.5 Communication non authentifiée malgré le protocole SSL

Lors de l'initiation de la session SSL, une fois que la plate-forme reçoit le certificat de la plate-forme où l'agent veut migrer, celle-ci vérifie que les attributs du certificat reçu sont conformes aux égalités fournies par l'agent en appelant la méthode *certCompare* qui rend un booléen selon que le certificat reçu est conforme ou non. Si c'est le cas, la plate-forme envoie l'agent, sinon elle termine la session SSL et renvoie l'agent sur la plate-forme de son propriétaire. La librairie IAIK-JCE de sécurité développée par IAIK (Institute for applied information processing and communications) permet d'accéder facilement à ces champs d'un certificat X.509. L'utilisation de cette librairie est gratuite pour la recherche. Nous n'avons pas pu mettre en œuvre ces solutions car le code source de Grasshopper n'est pas disponible. Cependant, l'existence de cette attaque sur notre implémentation ne modifie pas la validité des mesures qui ont été prises, car la solution à ces attaques modifie de manière non significative les

caractéristiques du comportement de l'agent en termes de temps d'exécution et de charge réseau utilisée par ce dernier.

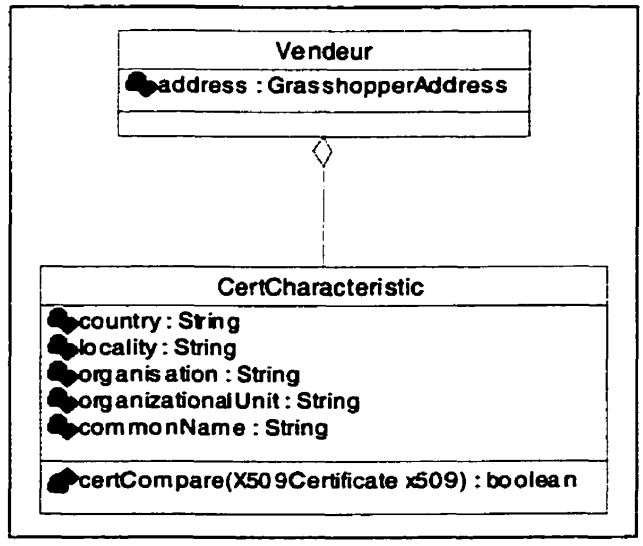

Figure 4.6 Classe CertCharacteristic

Bien entendu, en dehors de notre protocole, il n'est pas toujours nécessaire d'être absolument certain de l'identité de la plate-forme où l'agent est envoyé : on peut utiliser SSL pour la confidentialité des données et/ou pour l'intégrité de l'agent. C'est pourquoi l'utilisation d'une des deux solutions garantissant une authentification mutuelle doit être optionnelle.

Pour pouvoir utiliser le service de communication SSL, il faut une infrastructure à clé publique. Pour la génération des clés, nous avons utilisé l'outil *keytool* de Sun, fourni avec JDK (Java Development Toolkit). Le stockage des clés est effectué dans le *keystore*, nous avons utilisé l'implémentation de *keystore* de Sun, elle est aussi fournie avec JDK. Nous avons codé une autorité de certification qui, à partir des demandes de certificats générées par *keytool*, produit et signe des certificats pour les entités de notre application. Les librairies de Sun ne permettent pas de le faire, nous avons donc utilisé la librairie Java de l'IAIK. Nous avons supposé que les informations des certificats étaient à jour lors de leur utilisation. En effet, les autorités de certification émettent, à intervalle

de temps régulier, des listes de révocation de certificats (LRC) lorsque les informations de certains certificats ne sont plus valides alors que leur date de validité l'est encore.

H2 : Les plates-formes fournissent un canal de communication authentifié aux agents coopérants.

Pour cette communication authentifiée, nous aurions pu utiliser certains services de communication de Grasshopper. Ce service propose les protocoles suivants :

- CORBA IIOP (Common Object Request Broker Architecture – Internet Inter ORB Protocol) : il utilise un mécanisme conforme aux standards pour se connecter à un objet en utilisant le service de nom CORBA.
- MAF IIOP : ce protocole est une spécialisation de CORBA IIOP développée pour les interactions dans le cadre d'un système d'agents mobiles. Il a été décrit dans les standards MASIF et assure la connectivité entre des systèmes d'agents de différentes compagnies.
- RMI (Remote Method Invocation) : il permet à des objets Java d'invoquer des méthodes d'autres objets, s'exécutant sur une autre machine virtuelle Java.
- RMI au-dessus de SSL.
- Socket : ce type de communication est la plus robuste et la plus rapide car elle évite le surcoût d'un modèle d'objet distribué.
- Socket au-dessus de SSL.

Le service de communication fonctionne comme illustré à la Figure 4.7 : l'agent client construit un « proxy » sur l'agent qui joue le rôle de serveur. Quand le client veut invoquer un service du serveur, il appelle une méthode du « proxy » (Étape 1), qui va demander la localisation de l'agent au service de domaine des agences (Étape 2), puis il va invoquer la méthode de l'agent à distance via le service de communication (Étape 3). Ce dernier va communiquer l'invocation par un protocole supporté par les deux agences (Étape 4). Puis, la méthode va être invoquée localement sur l'agent serveur (Étape 5).

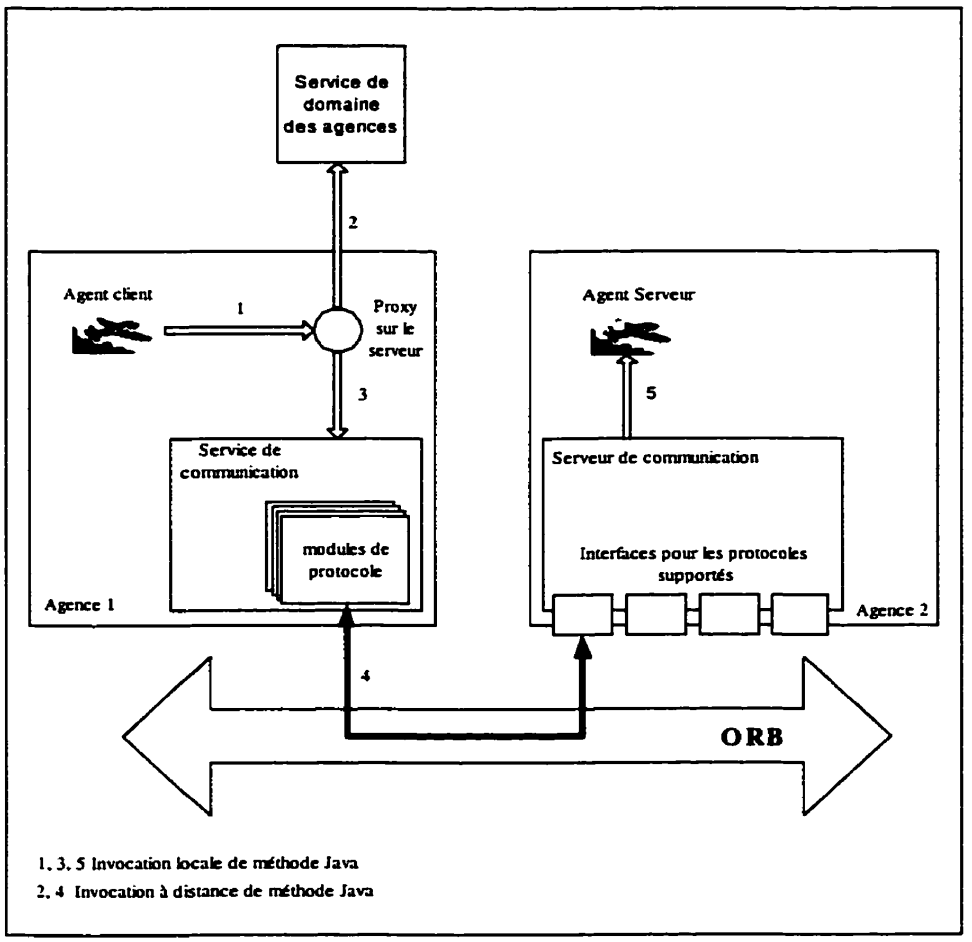

Figure 4.7 Communication inter agent sous Grasshopper

Il est possible de choisir un protocole de communication qui utilise SSL de manière à pouvoir forcer l'authentification mutuelle entre les deux agences. Il se pose comme auparavant le problème d'avoir accès à l'identité du serveur auquel le client s'est connecté. En effet, tel que mentionné précédemment, l'utilisation de SSL avec authentification du client et du serveur par Grasshopper est basée sur la vérification que le certificat du client ou du serveur est dans l'ensemble des certificats d'entités jugées de confiance par le propriétaire de la plate-forme. De plus, après la communication, il n'y a pas de méthodes pour que l'agent puisse obtenir le certificat du client ou du serveur avec lequel il y a eu une session SSL. Pour pouvoir utiliser ce service de Grasshopper, il

aurait fallu que le code source soit disponible, ainsi nous aurions pu rajouter des composantes pour notre utilisation.

Un autre point certainement plus important que celui que nous venons de détailler nous a confirmé dans l'idée de ne pas utiliser le service de communication de Grasshopper : lorsque le client demande à l'objet proxy d'invoquer la méthode du serveur, il contacte le service de domaine des agences pour obtenir l'adresse de l'agence où est l'agent serveur. Ce service de domaine est fourni par l'entité Région où sont enregistrées les agences avec leurs places respectives, ainsi que les agents qui sont sur ces places. Or, cette région doit être exécutée sur un ordinateur de confiance. En effet, si la région est contrôlée par un attaquant, les communications ne pourront être sécuritaires car les demandes d'adresse d'un serveur ne seront plus de confiance. Ceci crée un point faible pour l'implémentation de notre protocole. De plus, comme nous l'avons mentionné dans le chapitre III, l'un des intérêts de notre méthode d'enregistrement d'itinéraire est le fait qu'il n'est pas supposé que nous disposions d'un hôte de confiance sur le réseau. Or, si on utilise le service de communication de Grasshopper, pour fonctionner correctement, le protocole a besoin que la région utilisée soit exécutée sur un hôte de confiance.

Nous avons donc implémenté notre propre canal de communication authentifiée sans utiliser les services de communication de Grasshopper. Pour cela, nous avons utilisé la possibilité que Grasshopper offre de pouvoir associer à certaines places des services autres que ceux fournis par défaut. Ces services sont disponibles uniquement sur la place à laquelle ils sont associés. Nous avons créé deux services :

- l'un est appelé par l'agent *b*, il consiste à établir un serveur SSL qui écoute sur un port défini à l'avance (6000 dans l'implémentation). Lors d'une connexion, il reçoit des informations qu'il passe à l'agent lorsque la connexion est terminée.
- l'autre est appelée par l'agent *a* et consiste à se connecter à l'adresse donnée en paramètres de la méthode, sur le port prédéfini et d'envoyer les informations conformément au protocole. L'agent connaît l'adresse où est l'agent coopérant, car il possède une liste ordonnée avec l'ensemble des hôtes visités par l'agent *b*.

Le client et le serveur vérifie que le certificat est valide en comparant le nom de l'agence avec celui du certificat. Pour le client, il était simple d'obtenir le certificat de la plate-forme avec laquelle il s'était connecté et ainsi de pouvoir effectuer la vérification de la conformité de ce certificat. Pour le serveur, il n'était pas possible d'obtenir les certificats de clients qui se sont connectés. Pour faire la vérification de la validité des clients, nous avons dû redéfinir l'implémentation de l'interface X509TrustManager. Cette classe a notamment pour rôle d'accepter ou de rejeter la connexion d'un client ou d'un serveur en fonction du certificat utilisé dans cette dernière. L'implémentation de base accepte un certificat ou une chaîne de certificats qui a été signé par une autorité de certification dont la clé publique est définie comme étant de confiance dans le keystore associé à ce TrustManager. Pour implémenter notre propre TrustManager, nous avons notamment fait un ET logique entre la réponse faite par l'implémentation de base et notre vérification de conformité, i.e. le nom associé au certificat doit être le même que celui de la plate-forme d'où vient l'agent.

La Figure 4.8 et la Figure 4.9 montrent le diagramme de classe des services fournis par la place où les agents *a* et *b* sont exécutés. On voit notamment les méthodes pour que l'agent *a* se connecte à la plate-forme où est l'agent *b*. Dans le cas du client et du serveur, on utilise la clé privée et le certificat de la plate-forme pour initialiser le contexte de la session SSL. Il ne peut pas y avoir de vol d'informations secrètes (comme la clé privée) par l'agent, car le code est exécuté par la plate-forme et non comme s'il faisait partie de l'agent. Ce n'est pas l'agent qui possède le code pour initialiser la session SSL. Par exemple, il ne demande pas la clé privée pour répondre aux défis cryptographiques envoyés par l'autre entité lors de l'initialisation. On force l'authentification du client et du serveur dans la connexion SSL. Pour être sûr qu'on a bien ouvert une communication avec la plate-forme voulue, on vérifie, aussi bien au niveau de l'agent *a* qu'au niveau de l'agent *b*, que le certificat correspond à l'identité de la plate-forme avec laquelle on voulait communiquer. Dans un but de simplification, cette vérification consiste à comparer le nom de l'agence où on envoie l'agent et le nom de l'entité pour qui a été émis le certificat. Ainsi, si un vendeur X, qui a un certificat

signé par une autorité de certification de confiance, fait une attaque comme illustré à la Figure 4.5 lors d'une communication SSL entre un vendeur Y et un vendeur Z, l'initialisation se fera car le certificat est considéré comme étant de confiance; cependant, l'attaque sera détectée étant donné que le nom du certificat ne correspond pas à la plate-forme avec laquelle on voulait communiquer.

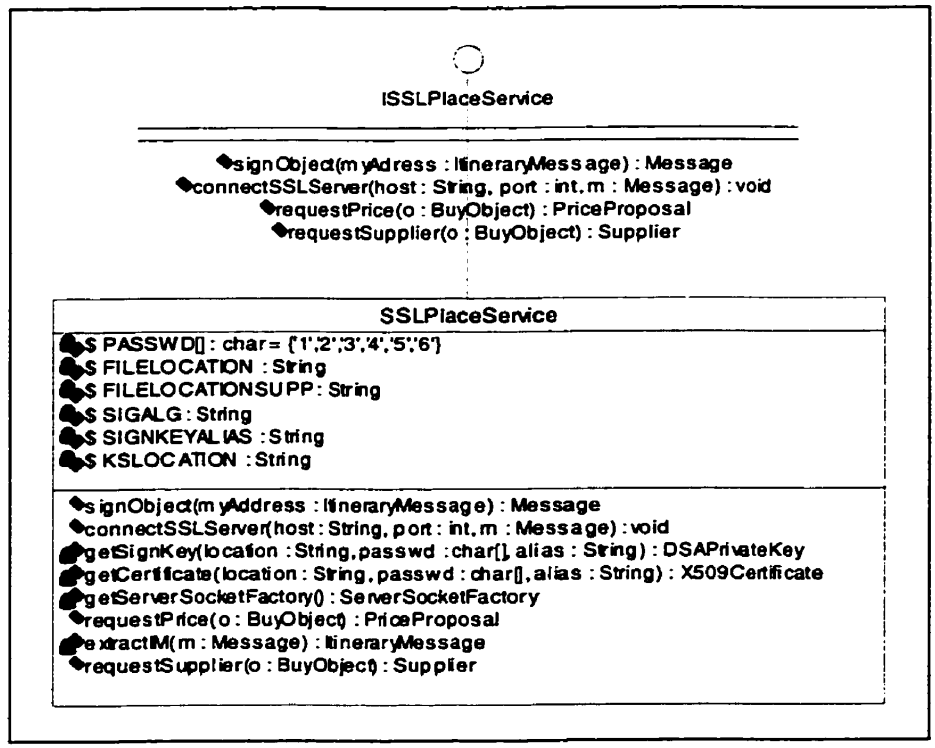

Figure 4.8 Diagramme de classe des services proposés par une place où est l'agent *a*

Nous avons envisagé de créer notre propre protocole pour la communication entre l'agent *a* et l'agent *b*. En effet, SSL assure la confidentialité des données et nous n'en avons pas besoin. Il était donc possible d'établir un protocole plus léger qui permettrait d'assurer l'authentification mutuelle et l'intégrité, ceci en utilisant une infrastructure à clés publiques et des défis cryptographiques. Cependant, SSL étant devenu un standard accepté par tous les acteurs d'Internet, il nous a semblé que

l'utilisation de ce protocole permet de prendre moins de risques quant à l'acceptation des propriétés de sécurité, même s'il a pour conséquence un léger surcoût.

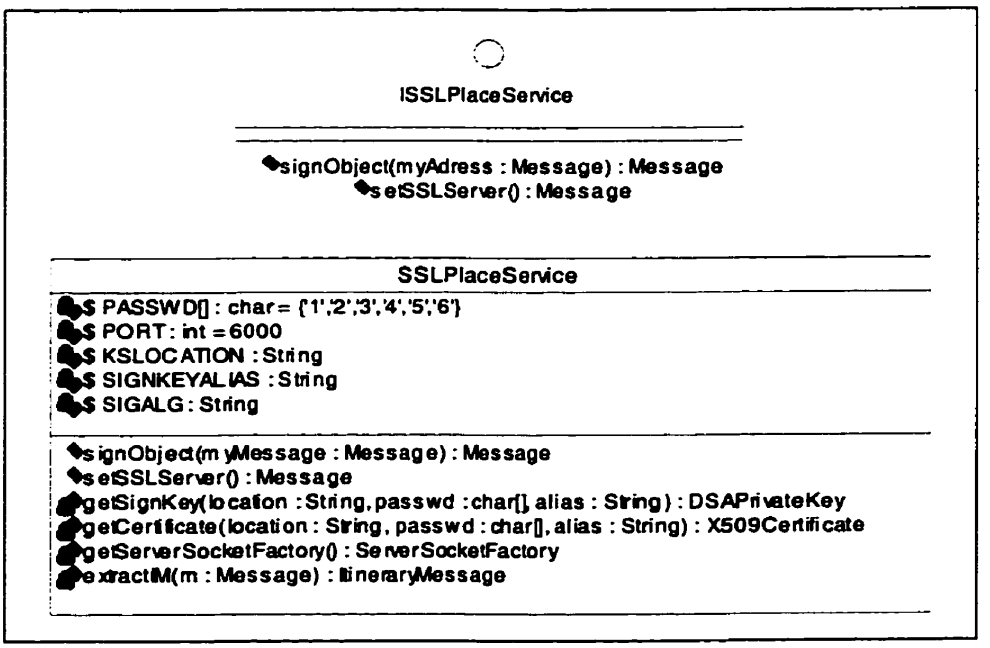

Figure 4.9 Diagramme de classe des services proposés par une place où est l'agent *b*

Pour coder la session SSL et son initialisation, nous avions le choix entre deux librairies :

- iSaSiLk développée par IAIK, cette librairie a été utilisée par les concepteurs de Grasshopper pour leur service de communication. Aussi est-elle gratuite pour des activités de recherche.
- JSSE (Java Secure Socket Extension) développée par Sun, elle est gratuite pour un usage non commercial.

Ces deux librairies implémentent la version 3.0 de SSL définie par Netscape. Nous avons choisi la librairie de Sun pour des raisons de compatibilité. En effet, tel que mentionné précédemment, nous avons utilisé des outils de Sun comme le *keytool* pour la génération de clés ou l'implémentation de SUN du *keystore* pour les stocker.

H3 : L'agent a accès, de manière authentifiée, à l'identité de la plate-forme distante, à celle de la plate-forme sur laquelle il s'exécute et à celle de la précédente où il s'était exécuté.

L'accès à l'identité de la plate-forme distante se fait par la vérification du certificat qui a été échangé lors de la communication SSL, comme nous l'avons expliqué précédemment. Si l'initialisation s'est déroulée sans erreur, c'est que la plate-forme distante possède la clé privée associée à ce certificat. Elle a bien l'identité qu'elle prétend avoir, car on suppose le vol de clé impossible. L'agent a accès de manière authentifiée à l'identité de la plate-forme sur lequel il s'exécute lorsqu'il vérifie la signature de l'objet *Message* que lui fournit la plate-forme. Par contre, l'accès a l'identité authentifié de la plate-forme précédente n'est pas possible de manière sûre. Cela est lié à l'utilisation de SSL faite par Grasshopper.

Implémentation du message échangé entre l'agent a et l'agent b

L'agent *a* construit un objet *ItineraryMessage* à partir des identités $prev a_i$, $id(ha_i)$, $next a_i$. Il appelle la méthode *signObject()* qui est un service fourni par la place où il se trouve. Cette méthode rend un objet Message qui contient le certificat de la plate-forme et un objet de type *SignedObject* incluant l'objet *ItineraryMessage* et la signature de la plate-forme. La Figure 4.10 montre le diagramme de classe des objets Message et *ItineraryMessage*. Cette méthode est délicate car une signature met en jeu la responsabilité de la personne (physique ou morale) qui signe. Nous avons donc fait effectuer la signature par le service de la place : il aurait aussi été possible de passer la clé privée de la plate-forme pour que ce soit l'agent qui effectue la signature. Mais, cette solution était dangeureuse car un agent malhonnête aurait alors pu signer au nom de la plate-forme. C'est aussi contre cette menace que nous avons restreint les paramètres de la méthode *signObject()* aux objets de type *ItineraryMessage* (méthode qui est appelé par l'agent *a*) et aux objets de type de *Message* (méthode est appelée par l'agent *b*). Avec les vérifications dynamiques de type effectuées par la machine virtuelle Java, il est impossible de faire signer un objet de type différent. L'agent *a* vérifie que la signature

est conforme à la clé publique du certificat. Si c'est le cas, elle envoie via le canal authenfié l'objet Message, qui est retourné à l'agent *b* par le service fourni par la place. Ce dernier va appeler la méthode *signObject()* pour obtenir la signature de la plate-forme hb_i. Ensuite, l'agent *b* vérifie la correspondance entre le certificat et la signature et que les propriétés du protocole sont respectées. Si c'est le cas, il ajoute cette entrée au vecteur d'enregistrement de l'itinéraire.

Les détails et les limites de notre implémentation étant connues, nous allons décrire les tests et les mesures qui ont été effectuées sur cette implémentation, ce qui fait l'objet de la prochaine section.

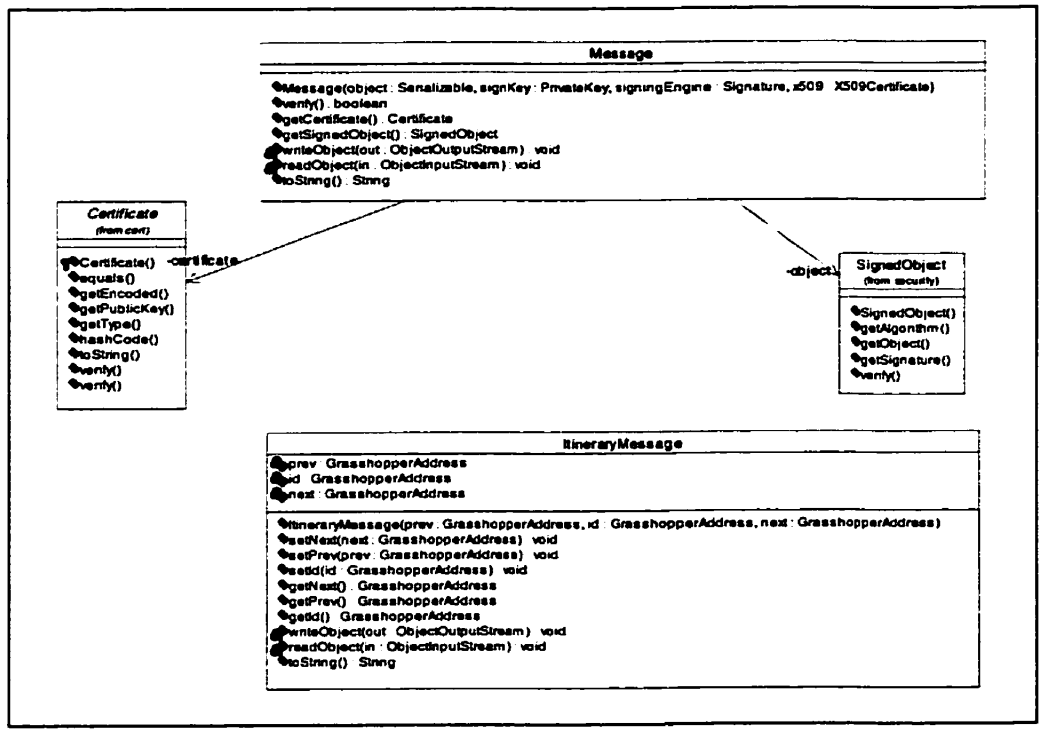

Figure 4.10 Diagramme des classes ItineraryMessage et Message

4.3 Tests et évaluation de notre implémentation

Pour faire nos tests, nous avons utilisé 5 machines dont les principales caractéristiques sont décrites dans le Tableau 4.1. Le nom des machines correspond au nom que nous avons utilisé pour l'application de magasinage. Le réseau utilisé est un réseau local de type Ethernet 100 Mbps.

Tableau 4.1 Description des machines utilisées pour les tests

Type	Nom de la plate-forme	RAM (Mo)	Système d'exploitation	Processeur
Ordinateur de bureau	Supplier1	192	Windows NT 4.0 Workstation	Intel Pentium II 400
Ordinateur de bureau	Supplier2	128	Windows NT 4.0 Workstation	Intel Pentium II 400
Ordinateur de bureau	B1	128	Windows NT 4.0 Workstation	Intel Pentium II 400
Ordinateur de bureau	B1	128	Windows NT 4.0 Workstation	Intel Pentium II 400
Portable	Home	256	Windows NT 4.0 Workstation	Intel Pentium III 400

4.3.1 Tests de notre implémentation

Nous avons tout d'abord testé la communication authentifiée pour le transport de l'agent dans les cas décrits au Tableau 4.2. Pour le client comme pour le serveur, nous avons testé trois cas :

- le client ou le serveur possède un certificat signé par une autorité de certification qui est considérée comme étant de confiance par l'autre entité;

- le client ou le serveur possède un certificat signé par une autorité de certification qui n'est pas considérée comme étant de confiance par l'autre entité;
- le client ou le serveur ne possède pas de certificat.

On voit que la connexion SSL est refusée lorsque le certificat du client ou du serveur n'est pas de confiance. Cela est conforme à nos hypothèses car nous avons forcé l'authentification mutuelle pour notre connexion SSL.

Tableau 4.2 Résultat des tests sur le transport des agents de manière authentifiée

Client	Serveur	Résultat
Certificat signé par une autorité de confiance	Certificat signé par une autorité de confiance	Transport de l'agent effectué
Certificat signé par une autorité de confiance	Certificat signé par une autorité qui n'est pas de confiance	Connexion SSL refusée : pas de transport de l'agent.
Certificat signé par une autorité de confiance	Pas de certificat	Connexion SSL refusée : pas de transport de l'agent.
Certificat signé par une autorité qui n'est pas de confiance	Certificat signé par une autorité de confiance	Connexion SSL refusée : pas de transport de l'agent.
Certificat signé par une autorité qui n'est pas de confiance	Certificat qui n'est pas signé par une autorité de confiance	Connexion SSL refusée : pas de transport de l'agent.
Certificat signé par une autorité qui n'est pas de confiance	Pas de certificat	Connexion SSL refusée : pas de transport de l'agent.
Pas de certificat	Certificat signé par une autorité de confiance	Connexion SSL refusée : pas de transport de l'agent.
Pas de certificat	Certificat signé par une autorité qui n'est pas de confiance	Connexion SSL refusée : pas de transport de l'agent.
Pas de certificat	Pas de certificat	Connexion SSL refusée : pas de transport de l'agent.

Nous avons ensuite testé la communication authentifiée entre les deux agents. Il faut, dans ce cas, rajouter la possibilité que l'une des deux entités communicantes

possède un certificat qui ne vérifie pas les conditions que nous avons fixées, c'est-à-dire que le nom associé au certificat de l'entité paire soit, pour l'agent client (l'agent *a*), le même que celui de l'agence où il veut communiquer et pour l'agent serveur (l'agent *b*), le même que celui où l'agent *a* devrait être. Le transport de l'agent ne se fait que si le client et le serveur possèdent un certificat qui est signé par une autorité de certification de confiance et que ce certificat vérifie les conditions définies auparavant. Dans les autres cas, la connexion SSL est refusée et le message n'est pas envoyé de l'agent *a* vers l'agent *b*. Le fait que la connexion SSL soit refusée même si une plate-forme possède un certificat signé par une autorité de confiance mais qui ne vérifie pas nos conditions de validité est normal, car nous avons choisi de redéfinir l'interface X509TrustManager qui décide si la communication avec l'entité paire est établie ou non. Il aurait pu en être autrement. En effet, nous avions, dans un premier temps, choisi de faire, pour le client, nos vérifications sur le certificat une fois que la communication était finie. Cette méthode avait pour avantage d'être plus simple à coder, mais la connexion était établie avec un serveur qui possédait un certificat signé par une autorité de certification de confiance et dont le certificat n'était pas forcément conforme à nos hypothèses. Une fois la communication terminée, l'agent *a* recevait le certificat et vérifiait s'il s'agissait de celui de la plate-forme avec qui il voulait communiquer. Un autre inconvénient de cette méthode demeure qu'elle n'est pas applicable du coté de la plate-forme serveur.

Enfin, nous avons testé la détection des attaques sur l'itinéraire. Nous avons effectué l'attaque où la plate-forme ha_i envoie l'agent *a* vers une plate-forme différente de celle où ce dernier veut aller. Nous l'avons effectuée grâce à une fonctionnalité de la plate-forme qui permet à l'utilisateur de forcer la migration d'un agent vers une certaine plate-forme. Nous avons aussi testé de cette manière la même attaque avec tentative de remise de l'agent sur sa route. Dans les deux cas, l'attaque a été détectée lors de la même étape i. Cependant, les tests que nous avons pu faire concernant les autres attaques sont limités quant aux propriétés de sécurité qu'elles vérifient. En effet, les attaques où il y a collaboration entre un hôte sur l'itinéraire de l'agent *a* et un hôte sur l'itinéraire de l'agent *b*, consistent en la manipulation de l'agent. Cette manipulation a pour but de

modifier le résultat d'une égalité ou de modifier la communication inter-agent. Ces attaques proviennent de la plate-forme qui s'exécute d'une manière non conforme par rapport à son comportement normal. Pour pouvoir effectuer ces tests, il nous aurait fallu faire de l'ingénierie inverse sur l'implémentation de la plate-forme pour comprendre son fonctionnement et remplacer un certain nombre de classes dans les packages de Grasshopper. Or ses compétences sont d'un niveau trop élevé pour que leur apprentissage puisse se faire dans un temps raisonnable dans le cadre de ce mémoire. La solution que nous avons retenue pour modéliser une attaque de ce genre de la part de la plate-forme est la modification du code de l'agent et la recompilation : par exemple, dans une attaque, la plate-forme sur laquelle s'exécute l'agent *b* le manipule pour qu'il croie qu'une égalité est vraie alors qu'elle est fausse. Pour tester cette attaque, nous avons modifié le code de l'agent *b* pour qu'il fasse comme s'il avait été manipulé, c'est-à-dire pour qu'il croie que l'égalité est vraie. Le Tableau 4.3 décrit les différentes attaques avec collaboration, la modélisation que nous avons choisie pour tester chacune des attaques, ainsi que le résultat du test et le moment de la détection. Nous avons effectué ses tests à la première étape car nous disposions d'un nombre de machines limité. Les résultats sont concluants puisque toutes les attaques testées sont détectées de manière conforme à notre protocole.

4.3.2 Évaluation de notre implémentation

Nous avons choisi de mesurer le coût de notre protocole sur une application proche d'une utilisation des agents mobiles dans un contexte réel. Il s'agit de l'agent de magasinage que nous avons décrit à la section 4.1. L'agent part avec une liste de trois produits qu'il doit magasiner. Il connaît l'adresse d'un seul vendeur, appelé *supplier1* (cf. Tableau 4.1 pour les caractéristiques de chacune des machines utilisées). Ce vendeur va le renvoyer vers un autre vendeur, car il y a un des produits qu'il ne vend pas et où il a des accords pour renvoyer les agents qui cherchent ce produit au deuxième vendeur, appelé *supplier2*. L'agent *a* possède la liste des hôtes que visitera l'agent *b*, qui contient les adresses des plates-formes b1 et b2. Nous avons choisi un nombre aussi peu élevé de

vendeurs par manque de machines. Il faut, en effet, rajouter une machine pour l'agent *b* en plus de celle du vendeur à chaque fois qu'on veut que l'agent de magasinage visite un vendeur de plus. Une possibilité pour pouvoir avoir un itinéraire de l'agent de magasinage plus long aurait été de faire exécuter plusieurs plates-formes sur une même machine. Cela est possible en faisant écouter les plates-formes sur des ports différents. Mais, dans ce cas, nos mesures de temps d'exécution aurait eu beaucoup moins d'intérêt, car elles n'auraient pas reflété une exécution normale.

Tableau 4.3 Attaques avec collaboration et modélisation pour les tests

Collaboration entre ha_i et hb_j pour :	**Attaque**	**Modélisation de l'attaque**	**Détection**	**Moment de la détection**
$i < j - 1$	Modification de l'enregistrement de l'itinéraire	L'agent *b* remplace le vecteur qui contient l'itinéraire par un autre	Oui	Retour de l'agent
$i = j - 1$	La plate-forme fait croire à l'agent que l'égalité $preva_{i+2} = id(ha_{i+1})$ est vraie	L'agent *b* ne vérifie pas cette égalité à cette étape	Oui	Étape i+2
$i = j$	La plate-forme modifie la communication entre *a* et *b*	L'agent *b* enregistre une valeur prédéfinie et ne tient pas compte du message envoyé par l'agent *a*	Oui	Étape i+1
$i = j - 1$ hb_{j+2}	Les deux plates-formes font croire que deux égalités sont vraies	L'agent *b* ne vérifie pas chacune des égalités à l'étape correspondante	Oui	Retour de l'agent

Nous avons mesuré :

- le temps d'exécution sur chaque machine de vendeur visitée par l'agent de magasinage;
- le temps total d'exécution de l'agent de magasinage et de son agent coopérant;
- une partie de la charge réseau utilisée pendant la tâche.

Nous avons choisi de prendre ces mesures car elles reflètent une partie des avantages du paradigme agent mobile par rapport à l'approche client-serveur. Il est donc important de mesurer l'influence de notre protocole sur ces paramètres. Les mesures de temps d'exécution ont été faites avec la méthode Java qui permet d'obtenir l'heure de l'horloge de la machine à la milliseconde près. Pour la première mesure, nous appelions cette méthode lors de l'arrivée de l'agent sur la plate-forme et avant qu'il ne migre vers la prochaine plate-forme. La deuxième mesure correspond à la différence entre l'heure de création des deux agents et le moment où les vérifications sur l'itinéraire effectuées par l'agent *b* lors du retour sur la plate-forme du propriétaire des agents sont terminées. Il faut ajouter que cette mesure correspond bien à la durée totale de la tâche car, sur la plate-forme du propriétaire, l'agent de magasinage avait fini sa tâche (afficher les résultats) avant l'agent *b*. Pour que les mesures sur le temps d'exécution soient fiables, nous n'avions qu'un seul processus qui s'exécutait sur notre machine : la plate-forme.
Nous avons effectué ces mesures sur trois types d'agents :

- l'agent de magasinage et son agent coopérant en utilisant notre protocole d'enregistrement d'itinéraire ;
- l'agent de magasinage et son agent coopérant en utilisant le protocole d'enregistrement d'itinéraire de Roth (1998) ;
- l'agent de magasinage seul .

Précisons que l'agent de magasinage a été testé en assurant le transport avec SSL, cela pour assurer son intégrité : le transport via socket ne permet pas d'assurer cette propriété qui est importante pour un agent de ce type. Lorsque les fonctionnalités de sécurité sont activées, par exemple pour assurer le transport des agents avec SSL, Grasshopper ajoute une signature de la partie fixe de l'agent avec la clé privée du propriétaire. À chaque fois que l'agent arrive sur une plate-forme, cette signature est vérifiée.

Le Figure 4.11 montre la taille en octets des différentes implémentations de ces agents. On voit que la taille des agents de notre protocole est plus élevée que celle des agents de Roth. Cela est normal car notre protocole est plus complexe avec l'emploi de

signature cryptographique notamment. Cependant, l'écart relatif entre la taille des classes de notre protocole et de celles de Roth n'est pas très important pour la classe ShoppingAgent (3%), mais il est plus important pour la classe Agentb (26%). Cela s'explique par le fait que les vérifications effectuées par l'agent *b* de notre protocole pour trouver le coupable en cas d'attaque sont plus complexes que celle faites par l'agent *b* du protocole de Roth dans la même situation. La classe de l'agent de magasinage seul est, elle, plus légère que les deux autres, respectivement, 19 % plus petit que la classe du protocole de Roth et 24 % par rapport à la classe de notre protocole. Cet agent ne communique pas avec un agent coopérant, ce qui explique sa taille plus faible.

Nous avons remarqué, lorsque nous avons pris les mesures, qu'il y avait de fortes variations, aussi bien au niveau de la charge réseau que du temps d'exécution, entre la première mesure et celles qui suivaient. Cela est dû à la machine virtuelle Java qui met en cache les classes qu'elle charge en mémoire. Lors de la prochaine instantiation de cette classe, la machine virtuelle ira chercher la classe dans sa cache et le temps pour instantier cette classe sera beaucoup plus rapide. Ainsi, si on modifie un agent, on le recompile et on l'envoie à une plate-forme à laquelle on a envoyé auparavant le même agent, l'agent qui sera exécuté sera celui qui est dans la cache de la machine virtuelle Java et non l'agent modifié. Nous avons donc effectué, pour chaque cas, une série de mesures avec, d'une part, une plate-forme qui n'avait jamais reçu les agents, i.e. une plate-forme dont la machine virtuelle n'avait pas en cache les classes envoyées et, d'autre part, une plate-forme qui avait déjà reçu l'agent, i.e. une plate-forme dont la machine virtuelle possède la classe dans sa cache. L'ensemble des mesures prises sur la plate-forme *supplier2* était très proche de celle de la plate-forme *supplier1*. Nous avons donc décidé de montrer uniquement celle prise sur cette dernière.

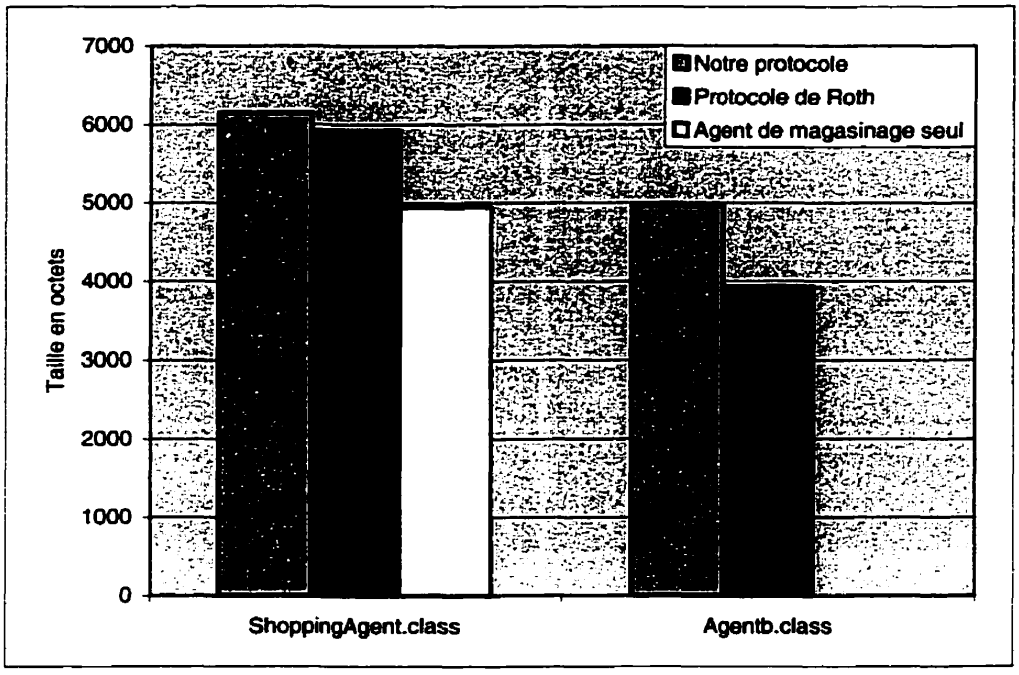

Figure 4.11 Taille des classes des deux agents pour les différents cas testés

La Figure 4.12 montre le temps d'exécution dans les trois cas étudiés, lorsque la plate-forme n'a jamais reçu l'agent. On note qu'il y a des variations assez importantes du temps d'exécution dans une catégorie donnée. Par exemple, la mesure 3 de notre protocole est beaucoup plus élevée que les autres mesures. Nous avons appris par les développeurs de Grasshopper que le « thread » qui exécute l'agent a une priorité minimum. Comme d'autres « threads » s'exécutent dans le processus plate-forme, il est possible que cela explique ces variations de temps d'exécution. De plus, le temps d'exécution ne diffère pas beaucoup entre notre protocole et celui de Roth (environ 7000 ms dans les deux cas), alors qu'il est beaucoup plus faible pour l'agent de magasinage. Cela s'explique par le fait que, dans notre protocole et celui de Roth, l'agent de magasinage établit une connexion SSL avec son agent coopérant, alors que l'agent de magasinage seul ne le fait pas. Or, établir cette connexion demande du temps car il faut que la machine virtuelle charge beaucoup de classes en mémoire et qu'elle initialise la session SSL avec l'entité paire.

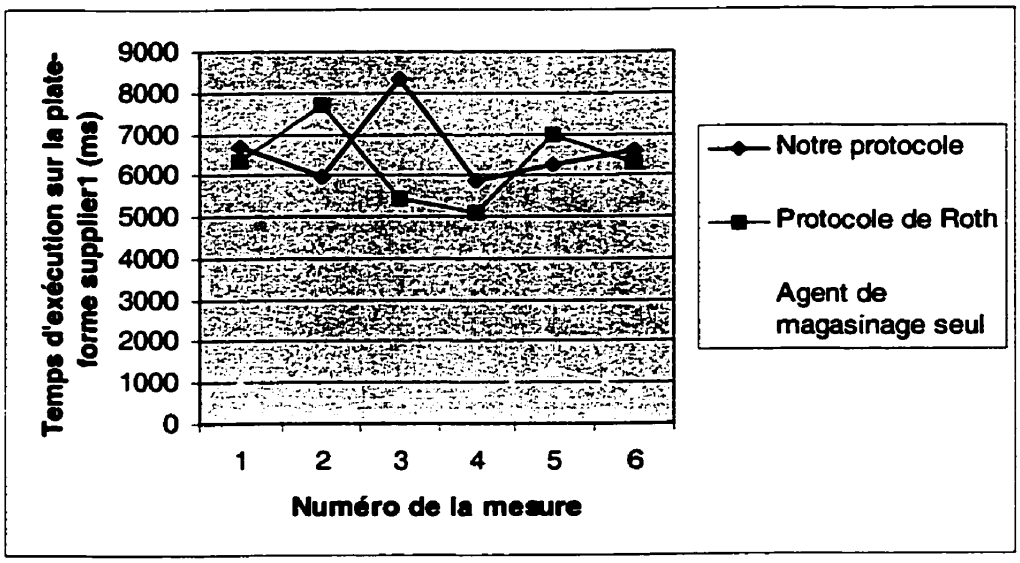

Figure 4.12 Temps d'exécution sur la plate-forme supplier1 sans cache

La Figure 4.13 montre le temps d'exécution total pour les trois cas testés lorsque les plates-formes n'ont jamais reçu les agents. De même que pour le temps d'exécution sur la plate-forme d'un vendeur, les mesures varient, notre protocole et celui de Roth ont des valeurs relativement proches et celui de l'agent de magasinage met beaucoup moins de temps à être exécuté. Ces résultats sont liés à ceux du temps d'exécution sur une plate-forme car le temps total d'exécution est égal à la somme des temps d'exécution sur chaque plate-forme plus la somme des temps des migrations. On peut supposer que le temps de migration varie peu d'une mesure à l'autre car les caractéristiques du réseau ne changent pas. De même, on peut supposer que la différence entre les temps de migration des trois agents est faible, car ils ont des tailles assez proches en regard du débit du réseau; et dans les trois cas, on utilise SSL pour transporter les agents. La Figure 4.14 confirme cette hypothèse. Les variations du temps total d'exécution s'expliquent donc par les variations du temps d'exécution sur chaque plate-forme et l'écart entre les valeurs des trois implémentations s'explique par l'écart dans le temps d'exécution sur chaque plate-forme visitée.

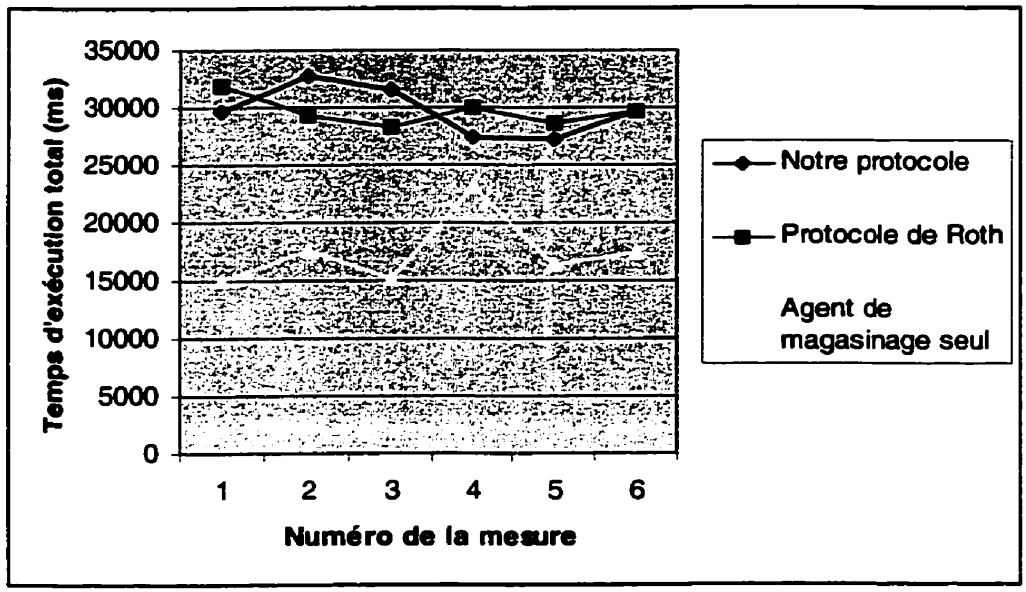

Figure 4.13 Temps total d'exécution sans cache

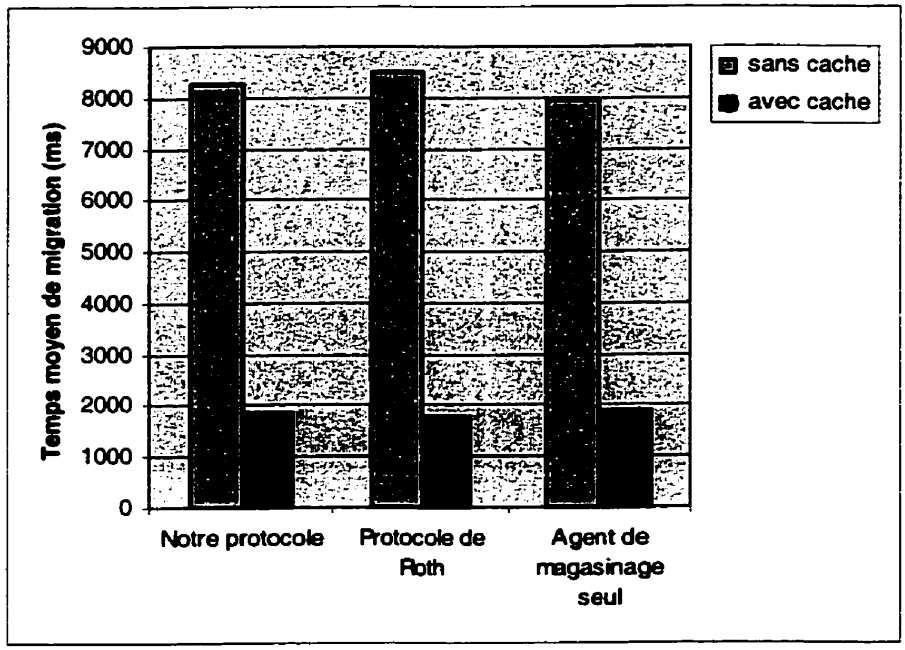

Figure 4.14 Temps moyen de migration

Les Figures 4.15 et 4.16 montrent respectivement le temps d'exécution sur la plate-forme *supplier1* et le temps total d'exécution lorsque la machine virtuelle Java a

les classes des agents et celles utilisées par ces derniers dans sa cache. La comparaison des trois cas donne les mêmes résultats que le cas sans cache : le temps d'exécution de notre protocole et celui de Roth sont assez proches alors que celui de l'agent de magasinage est beaucoup plus faible. Cela s'explique par le fait que, dans notre protocole et celui de Roth, il y a une communication SSL qui est établie avec l'agent coopérant et pas dans le cas de l'agent de magasinage seul. Notre protocole est plus long à s'exécuter que celui de Roth car il utilise des mécanismes cryptographiques que Roth n'utilise pas. Comme mentionné précédemment, le temps total d'exécution doit être égal à la somme du temps d'exécution sur chaque plate-forme plus le temps pris par les migrations. Or, ce dernier temps doit être petit car il n'y a pas besoin de transporter l'agent puisqu'il est dans la cache. Les variations entre les mesures dans le temps total d'exécution s'expliquent par les variations dans le temps d'exécution sur chaque plate-forme. Ces dernières s'expliquent par la valeur minimale de la priorité du « thread » qui exécute l'agent.

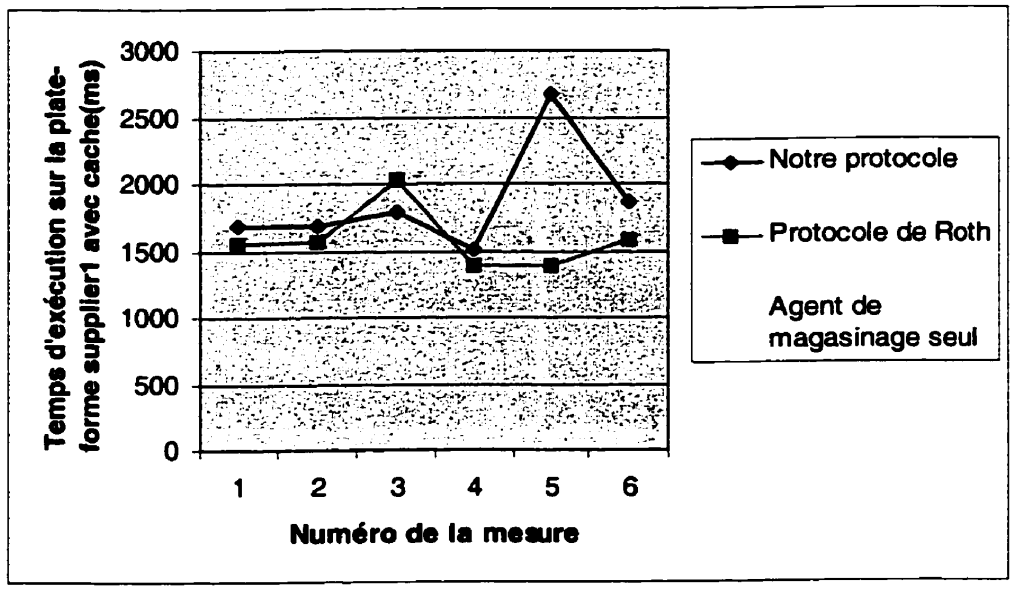

Figure 4.15 Temps d'exécution sur la plate-forme supplier1 avec cache

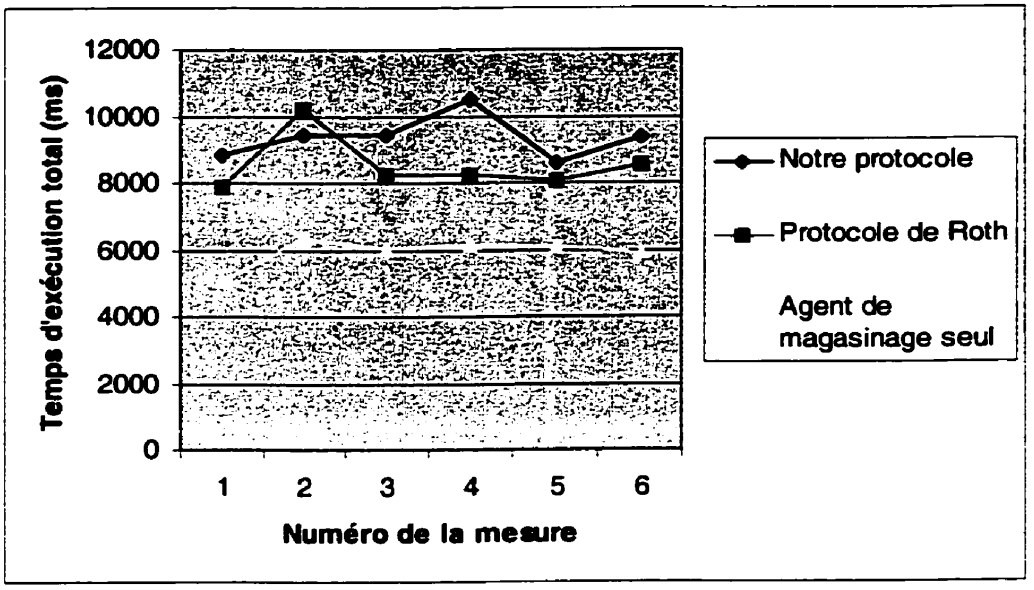

Figure 4.16 Temps total d'exécution avec cache

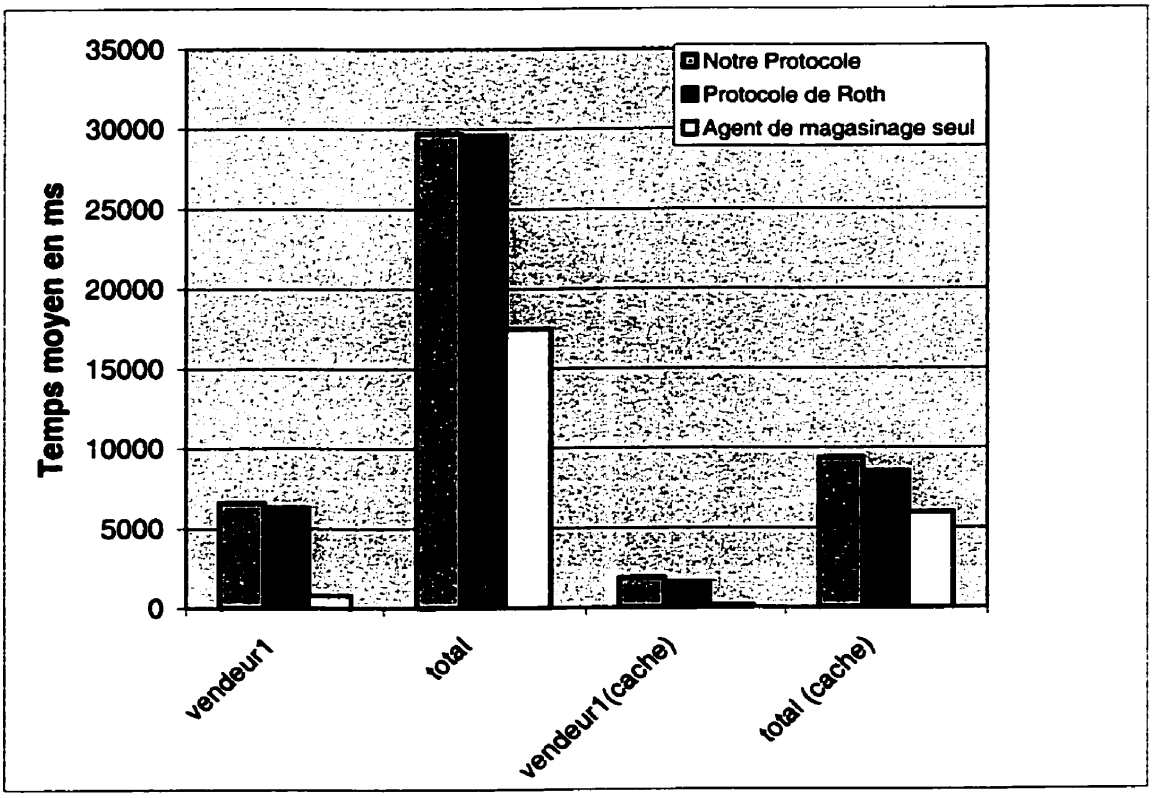

Figure 4.17 Temps moyen d'exécution dans les différents cas

La Figure 4.17 montre le temps moyen d'exécution dans les différents cas testés. Il ressort que notre protocole est, en moyenne, plus long à exécuter que les deux autres cas, au niveau de la plate-forme d'un vendeur ou au total. Cela se justifie par l'utilisation des signatures cryptographiques et de leurs vérifications, utilisées par l'agent *a* aussi bien que par l'agent *b* de notre implémentation. Cependant, selon le Tableau 4.4, l'écart relatif pour l'exécution sur la plate-forme d'un vendeur est quand même relativement faible par rapport au protocole de Roth et l'écart par rapport à l'agent de magasinage seul est très important. Ce faible écart s'explique probablement par la communication SSL entre les agents coopérants de notre protocole ou de celui de Roth. En effet, initier une session SSL est long car, en plus du temps d'établissement de la session et de la communication, il faut charger beaucoup de classes en mémoire pour la machine virtuelle. Donc, cela masque le temps pris par l'agent pour effectuer le reste de sa tâche. D'où la différence relative plutôt faible entre notre protocole et celui de Roth (4,7 %), alors que nous utilisons des mécanismes considérés comme coûteux, principalement la cryptographie asymétrique, que le protocole de Roth n'utilise pas.

On note aussi une forte différence entre l'écart relatif de notre protocole par rapport à l'agent de magasinage pour, d'une part, l'exécution sur la plate-forme d'un vendeur (694%) et, d'autre part, l'exécution totale (69%). Cela est probablement dû au temps de migration, car on a effectué la migration des agents (même l'agent de magasinage seul) avec SSL et, tel que mentionné précédemment, établir une connexion de ce type demande du temps. Même si l'agent de magasinage a un temps d'exécution sur chaque machine qui est beaucoup plus court que notre protocole, l'établissement de la communication et le temps de transport font que, au total, l'écart relatif de notre protocole par rapport à celui de Roth, pour le temps d'exécution sur la plate-forme d'un vendeur et le temps total, est plus faible, même si elle reste très importante.

Tableau 4.4 Comparaison des moyennes de temps d'exécution

écart relatif (en %)	Notre protocole / Protocole de Roth	Notre protocole / Agent de magasinage seul
Plate-forme vendeur	4,7	694,4
Total	0,5	69,4
Plate-forme vendeur (cache)	18,4	1172,3
Total (cache)	9,9	43,6

Avec cache, les mesures sur la plate-forme d'un vendeur sont, d'après nous, plus proche du temps UCT consommé par nos agents en tant que tel et non par le temps de chargement des classes. En effet, ce temps est faible car les classes sont dans la cache de la machine virtuelle. Dans ce cas, l'écart relatif entre notre protocole et celui de Roth est plus important que sans cache. Il en est de même pour l'écart relatif entre notre protocole et l'agent de magasinage seul. Cela est normal car, comme mentionné précédemment, notre protocole utilise des mécanismes coûteux en temps d'exécution et le temps de chargement des classes n'est plus là pour s'ajouter au temps total. L'écart relatif sur le temps total avec cache est plus petit entre notre protocole et l'agent de magasinage seul avec cache (43,6 %) que sans cache (69,4%). Le temps d'exécution sur chaque plate-forme est plus rapide avec cache, aussi bien pour notre protocole que pour l'agent de magasinage. Or, pour le transport de l'agent, il faut établir une connexion SSL à chaque migration. Cela prend du temps et cela masque le temps d'exécution qui est plus faible lorsque les classes sont dans la cache.

Nous avons mesuré la charge réseau grâce à un logiciel d'analyse de paquets, Iris, de la société eEye - Digital Security. Nous avons utilisé la version d'évaluation de ce logiciel. Nous avons filtré les paquets sur deux critères, les adresses IP de nos machines et les ports (6000 pour la communication inter-agent et 8000 pour le transport

des agents). Cependant, les machines qui exécutaient les plates-formes b1 et b2 n'étaient pas sur le même sous-réseau que la machine qui exécutait le logiciel d'analyse de paquets. Nous avons donc dû installer sur deux machines l'analyseur de paquets pour pouvoir mesurer tous les échanges. Là encore, nous avons séparé les mesures en deux parties : lorsque les classes n'étaient pas dans la cache de la machine virtuelle et lorsqu'elles le sont.

L'analyse des paquets révèle qu'il y a deux connexions dans le cas où les classes ne sont pas dans la cache de la machine virtuelle : l'une de la plate-forme qui envoie l'agent vers la prochaine plate-forme, puis l'autre depuis la prochaine plate-forme vers la plate-forme qui envoie l'agent. Lorsque les classes sont dans la cache, seule la première connexion est établie. On peut supposer que le client (la plate-forme qui envoie l'agent) envoie lors de la première connexion un message donnant le nom de la classe de l'agent à exécuter. Ensuite, la plate-forme qui doit recevoir l'agent demande à son *ClassLoader* (qui a été redéfini par les développeurs de Grasshopper) de mettre la classe en mémoire. Ce dernier cherche l'agent dans sa cache puis localement dans le *classpath*, et s'il ne l'a pas trouvé, il ouvre une connexion avec la plate-forme qui veut lui envoyer l'agent pour lui demander l'envoi de l'agent. Nous sommes certains que la plate-forme cherche d'abord localement l'agent qui est envoyé car il nous est arrivé qu'un mauvais agent soit exécuté par une plate-forme où nous avions envoyé l'agent parce qu'une classe de même nom était localement dans le *classpath*.

Pour les résultats de la Figure 4.18, nous avons ajouté la taille des paquets échangés entre le client et le serveur et réciproquement. Notons que le nombre d'octets échangés pour le transport de l'agent de magasinage (classe *ShoppingAgent.class*) de notre protocole est plus faible que celui du protocole de Roth. Nous ne nous expliquons pas ce résultat car la taille de la classe de notre implémentation est légèrement plus grande que celle de Roth. Une explication est l'utilisation possible d'algorithme de compression des données dans l'utilisation de SSL par Grasshopper. En effet, avant de chiffrer les données et de les transmettre, il est possible de les compresser. Cependant, du fait que le code source n'est pas public et que cette caractéristique n'est pas

documenté, nous ne savons pas si Grasshopper utilise cette compression lorsqu'il transporte un agent avec SSL. Par contre, le nombre d'octets échangés pour le transport de l'agent *b* est plus important pour notre protocole par rapport à celui de Roth, ce qui est conforme avec la taille des classes pour les deux implémentations (voir Figure 4.11 pour la taille des classes).

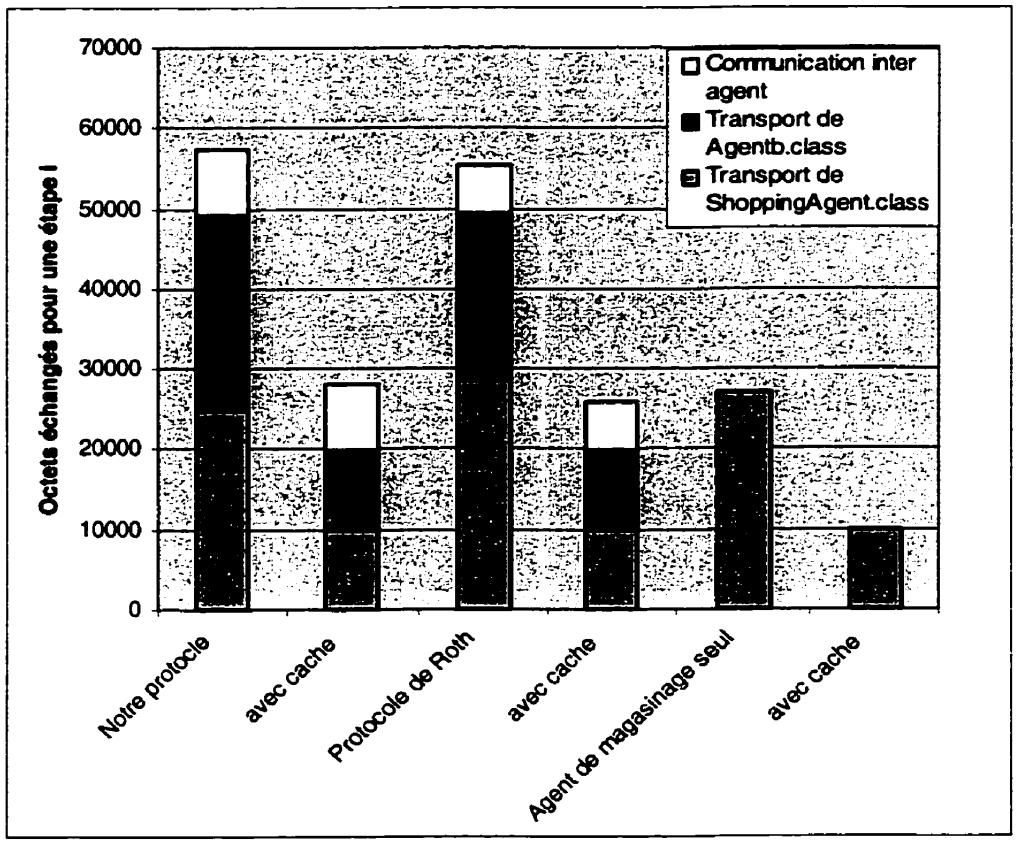

Figure 4.18 Octets échangés pour une étape i

On voit que le nombre d'octets échangés dans la communication inter-agent est le même pour les cas sans cache et avec cache, aussi bien pour notre protocole que pour celui de Roth, ce qui est normal car nous n'utilisons pas la possibilité qu'a une session SSL de reprendre après interruption, sans avoir à renégocier les paramètres de la session SSL entre le client et le serveur (algorithme de chiffrement et de code d'authentification

de message, taille de la clé ...). Donc, cela explique qu'on ait les mêmes résultats au niveau du nombre d'octets échangés avec ou sans cache. Le nombre d'octets échangés avec classes dans la cache est exactement le même pour les trois implémentations de la classe *ShoppingAgent.class* et très proche pour la classe *Agentb.class*. Nous ne savons pas expliquer de manière certaine ces résultats car le code source de la plate-forme n'est pas disponible et ce comportement n'est pas documenté. Il y a, selon nous, deux phénomènes qui peuvent rentrer en jeu. Le premier phénomène est constitué par le fait que la session SSL puisse être réutilisée sans avoir à renégocier tous ces paramètres, ce qui réduit le nombre de paquets échangés et donc, la quantité d'octets échangés. Le deuxième phénomène est que, comme la machine virtuelle possède la classe voulue dans son cache, elle ne demande probablement pas à la plate-forme précédente de lui envoyer la classe. Comme le nombre d'octets échangés est exactement le même pour les trois cas, on peut supposer que la classe n'est pas transmise car la taille des classes varie; et si cela avait été le cas, le nombre d'octets échangés aurait varié. Cela valide le fait que la classe de l'agent n'est pas transmise. En étudiant en détail les paquets échangés lorsque les classes sont en cache, on s'aperçoit que la connexion est la même au niveau des paquets échangés que celle qui a lieu entre la plate-forme où est l'agent et celle où il va être envoyé. Mais, comme mentionné précédemment, lorsque la classe de l'agent est dans la cache, il n'y a pas la deuxième connexion où la plate-forme qui envoie l'agent est serveur et celle qui va le recevoir est client. C'est cette communication qui, d'après nous, transporte le fichier de l'agent.

En ce qui concerne la communication inter-agent, le coût est plus important pour notre protocole avec un écart relatif de 36%. Cela s'explique par le fait que nous transmettons plus de données : alors que Roth ne transmet que les identités de trois plates-formes, nous transmettons les identités plus une signature cryptographique et le certificat X509 associé à cette signature.

Comme indiqué au Tableau 4.5, le nombre total d'octets échangés pour notre protocole et celui de Roth sont approximativement les mêmes, le nôtre est plus grand de 3,5 % pour le cas sans cache, 8,3 % dans le cas avec cache. L'écart relatif est plus grand

sans cache car la communication inter-agent est plus importante dans notre protocole que dans celui de Roth et le nombre total d'octets échangés est plus petit lorsque les classes sont en cache. Donc, le poids de la communication est plus grand avec cache. De la même manière, l'écart relatif entre notre protocole et l'agent de magasinage seul est plus grand avec cache. Les raisons sont les mêmes : le poids du coût de la communication inter-agent est plus important car le nombre total d'octets échangés au total est plus faible.

Tableau 4.5 Écart relatif du nombre d'octets échangés pour une étape

Écart relatif	Notre protocole / Protocole de Roth	Notre protocole / Agent de magasinage seul
Total sans cache	3,5	111,4
Total avec cache	8,3	182

CHAPITRE V
CONCLUSION

Ce chapitre présente une synthèse des travaux que nous avons effectués sur la protection de l'itinéraire d'un agent mobile. Nous verrons dans quelle mesure ces travaux ont été limités et par quels facteurs ils l'ont été. Enfin, nous donnerons des indications en vue de recherches futures.

5.1 Synthèse des travaux

Dans ce mémoire, nous avons présenté un protocole d'enregistrement d'itinéraire d'un agent mobile. Il s'agit d'enregistrer les identités des plates-formes qui ont exécuté un agent mobile, ceci de manière sûre. Ce problème est difficile à résoudre avec un agent seul car les données qu'il transporte sont susceptibles d'être manipulées par des plates-formes malveillantes. C'est pour cette raison que le concept d'agents coopérants a été introduit par Roth (1998).

Nous avons donc conçu un protocole d'enregistrement d'itinéraire en utilisant le concept d'agents coopérants. Nous avons décrit de manière précise le comportement de chacun des deux agents coopérants, ainsi que la procédure de décision pour établir les identités des coupables lorsqu'une attaque est détectée, ce qui n'avait pas été fait avec le protocole d'enregistrement proposé par Roth (1998). Pour démontrer les propriétés de sécurité de notre protocole, nous avons passé en revue l'ensemble des attaques possibles. Pour chacune, nous avons décrit le moment de la détection de l'attaque et, lorsque cela était possible, la désignation des coupables. Ainsi, nous avons pu vérifier que notre

protocole détectait les attaques où il y a collaboration entre une plate-forme appartenant à l'itinéraire d'un des deux agents coopérants et l'autre appartenant à l'itinéraire de l'autre agent. Ces attaques étaient supposées impossibles par le protocole d'enregistrement d'itinéraire de Roth et elles n'étaient pas détectées. La différence en terme de capacité de détection des deux protocoles correspond donc à ces attaques. En outre, nous avons quantifié la probabilité qu'il y ait une attaque non détectée par le protocole de Roth mais détectée par notre protocole détecte.

Enfin, nous avons implémenté notre protocole et celui de Roth sur une plate-forme d'agents mobiles. Nous avons choisi, pour tester ces implémentations, une application d'agent mobile de magasinage ayant un itinéraire non prédéfini. C'est une application classique utilisant le paradigme agent mobile où l'enregistrement d'itinéraire est critique. Nous avons testé notre protocole sur l'ensemble des attaques possibles et les résultats ont été conformes à nos prévisions en terme de désignation des coupables et en termes de moment de détection de l'attaque. Nous avons mesuré et comparé trois implémentations (l'agent de magasinage avec notre protocole, l'agent de magasinage avec le protocole de Roth et l'agent de magasinage seul). Les mesures que nous avons prises concernaient le temps d'exécution de l'agent de magasinage sur chaque plate-forme, le temps d'exécution total et la charge réseau utilisée. Ces mesures sont caractéristiques de certains des avantages des agents mobiles. Les résultats montrent que notre protocole a un coût qui est très légèrement supérieur par rapport à celui de Roth, aussi bien au niveau du temps d'exécution que de la charge réseau utilisée. Comme nous avons démontré que notre protocole détecte plus d'attaques que celui de Roth, il en résulte qu'il est plus intéressant d'utiliser notre protocole que celui de Roth. La différence de coût entre un agent de magasinage utilisant notre protocole et un agent de magasinage seul est assez grande aussi bien au niveau du temps d'exécution qu'au niveau de la charge réseau. On peut donc en déduire que notre protocole ne peut être utilisé dans des applications où ces coûts ont une importance critique. Il est à noter que, à notre connaissance, Roth n'a pas implémenté son protocole et n'a donc pas mesuré le coût de ce dernier.

5.2 Limitations des travaux

La première limitation concerne le rejeu des entrées de l'itinéraire. Notre protocole est, en partie, basée sur les signatures qui sont effectuées par les plates-formes où s'exécutent les deux agents coopérants. L'objet de ces signatures n'a pas de partie aléatoire, car il s'agit de trois identités de plate-forme. Par exemple, il est possible qu'une plate-forme qui exécute l'agent *b* garde en mémoire les signatures des plates-formes qui ont exécuté l'agent *a* et les rejoue lorsqu'un autre agent *b* s'exécute, mettant ainsi en jeu la responsabilité de la plate-forme dont il rejoue la signature.

Les principales autres limitations de nos travaux concernent l'implémentation de notre protocole et les tests que nous avons effectués. Nous nous sommes heurtés à plusieurs problèmes dont le manque de documentation précise sur le fonctionnement de la plate-forme et sur certaines librairies utilisées, principalement celle utilisée pour implémenter la communication inter-agent avec SSL.

De plus, certains problèmes n'ont pu être résolus du fait que le code source de la plate-forme n'est pas public, il n'est donc pas possible d'avoir accès à certaines composantes que nous aurions voulu utiliser. Ainsi, l'utilisation de SSL faite par la plate-forme que nous avons choisie pour mettre en œuvre notre protocole, Grasshopper, ne permet pas d'avoir une granularité assez fine en ce qui concerne l'acceptation ou le rejet de la communication SSL avec un client ou un serveur basé sur certaines caractéristiques de son certificat. Il nous a été impossible de modifier cette composante de Grasshopper du fait que le code source n'était pas public. Nous avons donc uniquement donné nos idées sur les solutions possibles à ce problème sans pouvoir les implémenter. Lorsque nous avons implémenté la communication inter-agent avec SSL, nous avons restreint les caractéristiques du certificat que l'entité paire devait avoir pour établir la communication dans un but de simplicité. De plus, nous avons utilisé une infrastructure à clé publique simplifiée puisqu'elle ne comportait pas de listes de révocation.

Les limitations sur les tests de notre implémentation viennent du fait que certaines attaques supposées sont impossibles sans faire de l'ingénierie inverse sur

l'implémentation de la plate-forme. En effet, lors de ces attaques, la plate-forme ne se comporte pas d'une manière normale. Nous avons trouvé une solution à ce problème en modifiant le code de l'agent mobile pour chaque attaque de manière à ce que son comportement soit le même que s'il avait été attaqué par la plate-forme. Cependant, cette solution n'est pas tout à fait satisfaisante car elle s'éloigne d'un cas d'attaque réelle. Elle nous a quand même permis de tester l'implémentation de nos algorithmes de désignation du coupable.

5.3 Indications des recherches futures

En ce qui concerne les recherches futures, il serait intéressant de trouver un algorithme qui permet de déterminer le coupable lorsque l'agent mobile est kidnappé : il s'agit du cas où le timer de l'agent coopérant qui attend que l'autre agent communique avec lui expire. Avec notre protocole, nous savons que le coupable est soit la plate-forme d'où a communiqué l'agent et qui l'a kidnappé avant sa migration, soit la plate-forme où devait aller l'agent mobile la dernière fois qu'il a communiqué avec l'agent coopérant. Nous ne savons pas si réaliser un algorithme pour déterminer laquelle des deux plates-formes est coupable est possible ou non.

Il faudrait chercher un mécanisme permettant d'éviter le rejeu des signatures. Une solution possible est de concaténer aux trois identités qui sont signées un nombre aléatoire qui serait propre à chaque agent. Ainsi, s'il y a une tentative de rejeu, elle sera détectée lors de la vérification des signatures car le nombre aléatoire ne correspondra pas à l'agent.

Il serait intéressant de réunir le travail que nous avons effectué avec celui de Karjoth (1998) pour concevoir une application de magasinage sécure. En effet, ce chercheur a donné un algorithme pour protéger les résultats intermédiaires d'un agent mobile contre la modification par les autres plates-formes visitées. Cependant, cet algorithme n'a des propriétés fortes que lorsque l'ensemble des plates-formes visitées est connu et fixé au départ de l'agent. L'algorithme n'a, par contre, pas besoin de savoir dans quel ordre les plates-formes sont visitées. Avec notre protocole d'enregistrement

d'itinéraire, il est possible d'avoir ces propriétés avec n'importe quel itinéraire, même s'il n'est pas prédéfini et s'il visite des plates-forme qui ne sont pas dans un ensemble de plates-forme connues dès le départ. En effet, si aucune attaque n'est détectée, on est sûr que l'agent s'est exécuté sur les plates-formes présentes dans notre enregistrement de l'itinéraire. Ainsi, on peut implémenter une application de magasinage à base d'agent mobile qui utilise tous les avantages de cette technologie, car nous ne sommes plus limités par un itinéraire qui doit être fixé dès le départ de l'agent.

BIBLIOGRAPHIE

R. A. Bazzi, "Code Hiding for Mobile Agents Security", Technical Report TR 1112998, Department of Computer Science and Engineering, Arizona State University, Novembre 1998.
http://www.larim.polymtl.ca/~gal/paper/outline.ps

S. Berkovits, J. D Guttman., V. Swarup, "Authentication for Mobile Agent", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne 1998, pp.114-136.

D. Chess, C. Harrisson, A. Kershenbaum, *Mobile Agents : Are they a good idea*, IBM Research Report RC 19887 (88465), Décembre 1994.
http://www.research.ibm.com/massive/mobag.ps

D. Chess, B. Grosof, C. Harisson, D. Levine, C. Parris, *Itinerant Agents for Mobile Computing*, IBM Research Report, 1995.
http://www.research.ibm.com/massive/rc20010.ps

D. Chess, "Security Issues in Mobile Code", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne, 1998, pp.1-15.

G. Cugola, C.Ghezzi, G. Picco, G.Vigna, "Analyzing Mobile Code Languages", in *Mobile Objet Systems : Towards the Programmable Internet*, Springer-Verlag, Berlin, Allemagne, 1997, pp. 93-111.

W. Farmer, J. Guttman, V. Swarup, "Security for Mobile Agents : Authentication and State Appraisal", *Proceedings of the 4th European Symposium on Research in Computer Security* (ESORICS'96), Rome, Italie, Septembre 1996, pp. 118-130.

W. Farmer, J. Guttman, and V. Swarup, "Security for Mobile Agents : Issues and Requirements", In *Proceedings of the 19th National Information Systems Security Conference*, Baltimore, Maryland., Octobre 1996, pp. 591-597.

J. Feigenbaum, P. Lee, "Secure Mobile-Code Applications". *DARPA Workshop on Foundations for Secure Mobile Code Workshop*, Mars 1997.
http://www.cs.nps.navy.mil/research/languages/statements/leefei.ps

A. Freier, P. Karlton, P. Kocher, "The SSL Protocol, Version 3.0", Internet Draft, Mars 1996.
http://home.netscape.com/eng/ssl3/ssl-toc.html

S. Goutet, "Mobile Agents Technologies : Implementation and Evaluation", Rapport interne, Ericsson, Avril 2000.
http://www.larim.polymtl.ca/~gal/paper/MA2.doc

J. Gosling, H. Mc Gilton, "The Java Language Environnement : A White Paper", Mai 1996.
http://java.sun.com/docs/white/languenv/index.html

M. S. Greenberg, J. C. Byington, T. Holding, D. G. Harper, " *Mobile Agents and Security*", IEEE Communications Magazine, juillet 1998, pp. 76-85.

F. Hohl "Time Limited Blackbox Security : Protecting Mobile Agents from Malicious Hosts", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne 1998, pp. 92-113.

F. Hohl, " A protocol to Detect Malicious Host Attacks by Using Reference States", Technical Report Nr., Faculty of Informatics, University of Stuttgart, Germany, août 1999.
ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-1999-09/TR-1999-09.ps.gz

F. Hohl, "A Framework to Protect Mobile Agents by Using References States", Proceedings of ICDCS 2000, 2000.
ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2000-03/TR-2000-03.ps.gz

W. A. Jansen "Countermeasures for Mobile Agents Security", Accepted for Publication in Computer Communications, Special Issue on Advances in Research and Application of Network Security, Elsevier Science BV, Été 2000.
http://www.itl.nist.gov/div893/staff/jansen/ppcounterMeas.pdf

W. A. Jansen, T. Kaygiannis, "Mobile Agent Security", NIST Special Publication 800-19, 1999.
http://csrc.nist.gov/mobileagents/publication/sp800-19.pdf

G. Karjoth, N. Asokan, and C. Gülcü, "Protecting the Computation Results of Free Roaming Agents", *Second International Workshop on Mobile Agents*, Stuttgart, Germany, Septembre 1998, pp. 195-207.

P. Lee, G. Necula, "Research on Proof-Carrying Code for Mobile-Code Security". *DARPA Workshop on Foundations for Secure Mobile Code Workshop*, Mars 1997. http://www.cs.nps.navy.mil/research/languages/statements/necula.ps

B. Lenou, "Services sur réseaux mobiles : Architectures et maintenance ", Mémoire de Maîtrise, École polytechnique de Montréal, Canada, Décembre 2000.

C. Meadows, "Detecting Attacks on Mobile Agents", *DARPA Workshop on Foundations for Secure Mobile Code Workshop*, Mars 1997. http://www.cs.nps.navy.mil/research/languages/statements/meadows.ps

G.C. Necula, P. Lee, " Safe, Untrusted Agents Using Proof-Carrying Code", *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne, 1998, pp. 61-91.

R. Oppliger, " Security Issues related to mobile code and agent-based systems", Computer Communications, 22 (1999), pp. 1165-1170.

J. J. Ordille, "When agents roam, who can trust ?", *Proceedings of the First Conference on Emerging Technologies and Applications in Communications*, Portland, Oregon, Mai 1996. http://cm.bell-labs.com/cm/cs/doc/96/5-09.ps.gz

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

J. Riordan, B. Schneier, "Environmental Key Generation Towards Clueless Agents", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne 1998, pp 15-24.

R. Rivest, A. Shamir, L. Adelman, "On Digital Signature and Public Key Cryptosystems", MIT Laboratory for Computer Science Technical Memorandum 82, 1977.

R. Rivest, A. Shamir, L. Adelman, "Cryptographic Communications System and Method", US Patent 4, 405,829, 1983.

R. Rivest, Internet RFC 1321, 1992.
http://theroy .lcs.mit.edu/~rivest/Rivest-MD5.txt

V. Roth, "Mutual Protection of Co-operating Agents", in *Secure Internet Programming*, Vitek et Jensen (Ed.), Springer-Verlag, Berlin, Allemagne, 1998, pp 26-37.

V. Roth, "Secure Recording of Itineraries Through Cooperating Agents", *Proceedings of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems : Secure Internet Mobile Computations*, pp 147-154, INRIA, France, 1998.

T. Sander, C. F. Tschudin, "Protecting Mobile Agents Against Malicious Host", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne 1998, pp 44-60.

F.B. Schneider, "Towards Fault-tolerant and Secure Agentry", in *Distributed Algorithms*, Mavronicolas et Tsigas (Ed.), Springer-Verlag, Berlin, Allemagne 1997, pp. 1-15.

C. Tschudin, "Mobile Agent Security", Chapitre 18 of M. Klusch (Ed.), in *Intelligent Information Agents - Agent based information discovery and management on the Internet*. Springer-Verlag, Berlin, Allemagne, 1999, pp. 431-445.

G. Vigna, "Protecting Mobile Agents through Tracing", *Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems*, Jyvälskylä, Finland, June 1997. http://cuiwww.unige.ch/~ecoopws/ws97/papers/vigna.ps.gz

G. Vigna "Cryptographic Traces for Mobile Agents", in *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, Berlin, Allemagne 1998, pp 137-153.

U. G. Wilhem, S. Staamann, L. Buttyan, " Introducing Third Parties to the Mobile Agent Pradigm", in *Secure Internet Programming*, Vitek et Jensen (Eds.), Springer-Verlag, Berlin, Allemagne, 1998, pp 469-492.

Bennet S. Yee, *A Santuary for Mobile Agents* , Technical Report CS97-537, University of California in San Diego, Avril 1997.

Jukka Ylitalo, "Secure Platforms for Mobile Agents", unpublished paper, janvier 2000 http://www.hut.fi/~jylitalo/seminar99/index.html

A. Young, M. Yung, "Slinding Encryption : A Cryptographic Tool for Mobile Agents", *Proceedings of the 4th International Workshop on Fast Software Encryption*, FSE'97, Biham (Ed.), Springer-Verlag, Berlin, Allemagne, janvier 1997, pp 230-241.