Contrôle d’Accès dans un Système à Agents Mobiles sur Java

D. Hagimont, L. Ismail

Projet SIRAC (IMAG-INRIA)

INRIA, 655 av. de l’Europe, 38330 Montbonnot Saint-Martin, France

Internet: {Daniel.Hagimont, Leila.Ismail}@imag.fr

Résumé: Cet article présente un modèle de protection pour des agents mobiles gérés sur l’environnement Java.

Dans ce modèle, le contrôle de l’accès aux objets est réalisé à l’aide de capacités logicielles pouvant être échangées entre des agents mutuellement méfiants. Chaque agent définit sa propre politique de protection, c’est à dire les règles de contrôle d’accès qui doivent être mises en œuvre lorsqu’il interagit avec d’autres agents.

Le principal avantage de l’approche proposée dans cet article est que la définition de la politique de protection d’un agent est séparée du code exécuté par cet agent. Cette définition de la protection est décrite à l’aide d’un langage de définition d’interface (IDL) au niveau de l’interface de l’agent, ce qui assure une plus grande modularité et facilite l’expression de la protection.

Un prototype de ce modèle de protection a été réalisé sur Java et les expérimentations à l’aide d’applications simples ont montré la faisabilité et les avantages de cette approche.

1 Introduction

La protection est un aspect fondamental de l’informatique répartie, en particulier lorsque des usagers coopèrent en partageant des applications ou des objets. Le développement d’applications réparties sur le réseau Internet a accru les besoins en termes de protection : l’ouverture sur Internet ne doit pas mettre en périls les machines connectées.

Plus récemment, la recherche en systèmes répartis a vu l’émergence d’un nouveau modèle pour la structuration d’applications réparties sur Internet : la programmation par agents mobiles [KGR96]. Dans ce modèle, un agent est un processus possédant un contexte d’exécution propre, incluant du code et des données, pouvant se déplacer de machine en machine (appelées serveurs) afin de réaliser la tâche qui lui est assignée. Généralement, un agent réalise des appels de méthode sur des objets exportés par les serveurs visités ou par des

agents rencontrés sur ces serveurs. Dans le contexte de la programmation par agents mobiles, la protection est devenue un des problèmes les plus importants, car elle est une condition stricte à l'acceptation de ce nouveau modèle : qui accepterait de l'utiliser si un agent peut contourner les mécanismes de protection et corrompre les données gérées par des serveurs ou des agents s'exécutant sur ces serveurs ?

Java [GM95] est probablement l'exemple le plus connu d'environnement fournissant des mécanismes pour gérer du code mobile. Java permet de développer des applications mobiles à partir d'un langage orienté-objet, ces applications pouvant se déplacer sur des machines hétérogènes. Du point de vue de la protection, l'avantage principal de Java est que son langage de programmation est un langage sûr ; le langage Java ne permet pas l'utilisation d'adresses virtuelles. Cette caractéristique du langage est la technique couramment utilisée pour assurer des fonctions de protection dans les systèmes à agents mobiles, mais cette technique ne permet pas aux agents et serveurs interagissant de contrôler de façon flexible les droits d'accès accordés aux autres agents et serveurs.

Dans cet article, nous proposons un modèle de protection basé sur des capacités logicielles, permettant à des agents de contrôler les droits d'accès accordés aux autres agents. Ce modèle de protection cumule les avantages suivants :

- Dynamicité. Le modèle de protection permet aux agents d'échanger des droits d'accès dynamiquement.
- Autonomie. Chaque agent est responsable de la définition de sa propre politique de protection, qui est liée à l'agent et déplacée avec l'agent.
- Suspicion mutuelle. Tous les agents sont égaux au regard de la protection. Le système n'impose pas un hiérarchie entre les agents coopérants.
- Modularité. La définition de la protection est séparée du code exécuté par l'agent, ce qui assure une plus grande modularité et facilite la définition de la politique de protection de l'agent.
- Portabilité. Nous pensons que l'intégration à un autre environnement pour agents mobiles est tout à fait réaliste.

Le modèle de protection pour agents mobiles que nous proposons a été réalisé sur Java. Bien que nous n'ayons pas réalisé un environnement pour agents mobiles dans sa totalité, nous avons implanté tous les mécanismes nécessaires à la validation du modèle. Nous avons également réalisé une application à base d'agents permettant d'illustrer l'utilisation du modèle de protection.

Le reste de l'article est structuré de la façon suivante. Dans la section 2, nous présentons les besoins auxquels le modèle de protection doit répondre. La section 3 présente les aspects de Java qui sont nécessaires à la compréhension de cet article. Le modèle de protection est présenté dans la section 4 et son implantation sur Java est détaillée en section 5. Après une comparaison avec des travaux similaires en section 6, nous concluons cet article en section 7.

2 Motivations et Besoins

Nous ne revenons sur l'intérêt de la programmation par agents mobiles dont de nombreux avantages sont discutés dans [KGR96]. Dans cette section, nous insistons sur les besoins en termes de protection des applications à base d'agents.

Pour assurer la protection des agents dans les systèmes à agents mobiles, trois problèmes importants doivent être pris en compte :

- Isolation. L'isolation prend ici le sens de confinement de l'exécution. Lorsqu'un agent est exécuté sur un serveur, le serveur doit garantir que l'agent ne peut accéder qu'à ses propres objets ou à des objets dont il a reçu explicitement des droits. L'agent ne doit pas être en mesure de contourner cette règle et d'adresser directement des objets sur lesquels il n'a aucun droit. Cette isolation est généralement fournie par le biais d'un langage sûr (comme Java) qui ne permet pas de manipuler des adresses virtuelles. Ainsi, un agent ne peut appeler un objet que s'il a explicitement reçu une référence à cet objet.
- Contrôle d'accès. Le contrôle d'accès est la définition de la politique qui détermine quand les agents[1] peuvent interagir. Un agent doit pouvoir exporter des références sur ses objets en contrôlant les droits qu'il donne sur ses objets. La politique de protection détermine quand et comment ces droits sont exportés.
- Authentification. L'authentification signifie ici la possibilité d'associer une identité à chaque agent. Cette identité peut être une identité d'usager, de groupe ou de machine. Quelle qu'elle soit, elle est ensuite utilisée pour fournir un environnement de protection initial à un agent arrivant sur un site. Par la suite, cet environnement de protection peut évoluer dynamiquement en fonction de l'exécution de l'agent.

Comme cela a été souligné auparavant, le premier problème est résolu par l'utilisation d'un langage sûr comme Java. L'authentification n'est pas le problème sur lequel nous nous sommes concentrés. Nous nous sommes attachés à fournir aux agents un moyen de contrôler facilement les droits accordés aux autres agents. Dans la suite, nous utilisons le terme *protection* pour parler du contrôle d'accès aux objets.

En concevant ce modèle de protection, nous avons voulu répondre aux besoins suivants :

- Dynamicité. Dans un système à agents, il est très important de permettre une évolution dynamique des droits des agents au cours de leur exécution. Lorsqu'un agent arrive sur un serveur, il s'exécute souvent dans un environnement de droits restreints, puis ses droits évoluent en fonction de son exécution et des agents qu'il rencontre.
- Autonomie. Chaque agent est responsable de la définition de sa propre politique de protection. Lorsque cette politique est définie par un programmeur, le programmeur

---

[1] Le serveur hôte peut être assimilé à un agent.

ne sait pas encore quels serveurs l'agent va visiter, ce qui rend impossible l'enregistrement de ces règles dans une couche système ou un serveur de protection centralisé. Les règles de protection d'un agent doivent être liées à l'agent et déplacées avec l'agent.

- Suspicion mutuelle. Tous les agents sont égaux au regard de la protection. Le système ne doit pas imposer une hiérarchie entre les agents coopérants comme cela est le cas dans les systèmes de protection basés sur un modèle maître/esclave.
- Modularité. Lors de la conception, nous n'avons pas voulu fournir une extension devant être utilisée depuis le langage de programmation (cela aurait rendu les programmes bien trop complexes). Nous avons donc séparé la définition de la protection du code exécuté par l'agent, ce qui assure une plus grande modularité et facilite la définition de la politique de protection de l'agent.
- Portabilité. Le modèle de protection que nous avons défini est suffisamment général pour être intégré dans des systèmes à agents différents. De plus, notre réalisation ne dépend pas fortement d'éléments propres à l'environnement Java. Nous pensons que l'intégration à un autre environnement pour agents mobiles est tout à fait réaliste.

3 L'Environnement Java

Dans cette section, nous présentons tout d'abord les aspects de Java relatifs à ce travail, puis nous donnons les principes de base de gestion d'agents mobiles sur Java.

3.1 Aspects de Java

Java est un langage orienté-objet similaire à C++ (d'un point de vue syntaxique) qui est utilisé pour produire des programmes pouvant être exécutés sur une machine virtuelle pratiquement disponible sur tous les types de machine. Etant donné que Java est très connu, nous insistons sur les fonctions que nous avons utilisées dans notre expérimentation : la mobilité du code, le polymorphisme, et la liaison dynamique.

Un aspect très important de Java est la mobilité du code. Java permet de charger dynamiquement des classes provenant d'un site distant. La mobilité du code requiert que le code soit portable et qu'il soit isolé afin d'éviter la propagation d'erreur. La portabilité du code est assurée par interprétation du code obtenu après compilation. En effet, le compilateur Java ne génère pas du code machine, mais un code indépendant de la machine qui est interprété par la machine virtuelle Java. L'isolation est assurée par le fait que le langage Java est sûr. Le langage Java ne donne pas directement accès à l'espace d'adressage du programme (les objets ne sont pas manipulés à travers des pointeurs, mais à l'aide de références gérées par le langage) et il est fortement typé, implantant des vérifications à la compilation et à l'exécution. En particulier, pour les classes chargées dynamiquement, Java vérifie que le code chargé est effectivement du code généré par la compilateur Java et il vérifie à l'exécution la conformité entre l'appelant et l'appelé lors de chaque appel de méthode. Il est également à

noter que Java fournit dans sa dernière version [Sun96] un outil de sérialisation d'objet qui permet de transporter des instances entre des environnements Java différents.

Un autre aspect de Java qui nous intéresse est le polymorphisme. Le polymorphisme est la possibilité de définir des interfaces (ou types) séparément des classes. Une interface est une définition de la signature d'un ensemble de méthodes, indépendamment de toute implantation. Une interface peut alors être réalisée par plusieurs classes et il est possible de déclarer une variable dont le type est une interface et qui peut référencer des objets de différentes classes qui sont des réalisations de l'interface de cette variable.

Enfin, le dernier aspect très important de Java est la liaison dynamique. Liaison dynamique signifie ici la possibilité de déterminer à l'exécution le code devant être exécuté lors d'un appel de méthode. Etant donné que Java permet de charger dynamiquement des classes, une variable dont le type est une interface peut être affectée pour référencer un objet dont la classe a été dynamiquement chargée. Java retarde la liaison du code à exécuter (à partir de cette variable) jusqu'à l'exécution, permettant d'exécuter une classe chargée dynamiquement.

3.2 Agents sur Java

Trois problèmes doivent être résolus pour gérer des agents mobiles sur un environnement comme Java :

- Exécution des agents. Une machine qui accueille des agents mobiles doit fournir des mécanismes permettant d'exécuter ces agents de façon concurrente. Java fournit la possibilité d'exécuter plusieurs processus légers (*threads*) dans un environnement Java qui s'exécute dans un processus Unix standard. Ainsi, on peut gérer l'exécution de plusieurs agents dans le même environnement. Chaque agent se voit associer un processus léger qui exécute le programme de l'agent.
- Déplacement des agents entre des environnements différents. Le second problème est de permettre le transfert d'un agent d'un site à un autre. Pour déplacer un agent, on doit être capable de transférer son code et ses données. Le code est composé de classes et les données d'instances de ces classes. Pour le transfert du code, Java fournit la possibilité de charger des classes dynamiquement, ces classes pouvant être locales comme distantes. Un serveur accueillant un agent peut donc charger dynamiquement les classes qui composent le programme de l'agent. Le mécanisme de sérialisation fournit par Java permet symétriquement de transférer des instances d'un environnement à un autre. Dans le cadre de notre travail, nous ne nous sommes pas concentré sur cet aspect du problème. Bien que nous ayons réalisé une partie de ces mécanismes, nous avons plus précisément étudié la gestion des droits d'accès dans un environnement à agents mobiles.
- Le partage d'objets. Le partage d'objets entre des agents qui s'exécutent dans le même environnement Java est implicite, car tous ces objets sont gérés dans le même espace d'adressage dans lequel l'environnement Java s'exécute. Cependant, pour

partager des objets, des agents doivent s'échanger des références d'objets (des pointeurs d'objets à la Java). Pour ce faire, l'environnement doit fournir un service de nommage permettant de publier des référence d'objets, c'est à dire d'associer des noms symboliques à des références d'objets et de retrouver une référence à partir d'un nom. Nous supposons que ce service de nommage est utilisé par les agents pour démarrer leur coopération. Lorsqu'un agent récupère une référence à un objet dans le serveur de nom, l'agent peut appeler cet objet en utilisant l'interface Java que cet objet est supposé réaliser.

4 Modèle de Protection

Dans cette section, nous présentons notre modèle de protection basé sur des capacités logicielles.

4.1 Un Modèle à Base de Capacités

Le modèle que nous proposons est basé sur des capacités logicielles [Levy84]. Le modèle à capacité a l'avantage de faciliter l'échange de droits d'accès à l'exécution, ce qui est un de nos objectifs (dynamicité).

Une capacité est un jeton qui identifie un objet et qui définit des droits d'accès sur cet objet, c'est à dire le sous ensemble des méthodes qui peuvent être appelées sur cet objet. Pour appeler un objet, un agent doit posséder une capacité sur cet objet autorisant cet appel. Lorsqu'un objet est créé, une capacité est retournée au créateur et contient les droits maximaux sur l'objet créé. Cette capacité peut être utilisée pour accéder à l'objet, mais elle peut également être copiée et transmise à un autre agent. Lorsqu'une capacité est copiée, ses droits peuvent être restreints afin de limiter les droits accordés à un autre agent.

Chaque agent s'exécute donc dans un environnement de protection dans lequel il a accès à tous les objets qui lui appartiennent. Le transfert de capacité d'un agent à un autre est réalisé lors d'un appel de méthode. Lorsqu'une référence d'objet est passée en paramètre, ce passage de référence peut s'accompagner d'un passage de capacité.

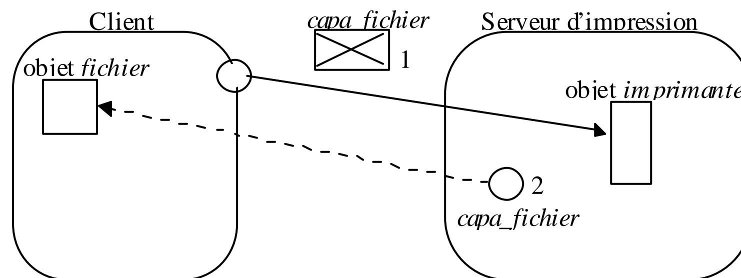

Figure 1. Serveur d'impression

Afin d'illustrer ce modèle à capacités, considérons l'exemple d'un agent serveur d'impression qui exporte un objet *imprimante* permettant d'imprimer un fichier (Figure 1).

Une capacité sur l'objet *imprimante* est donnée aux agents clients, leur permettant d'imprimer un fichier. Lorsqu'un client veut imprimer un fichier, l'objet *imprimante* doit recevoir le droit de lire le fichier à imprimer. Le client passe donc, lors de l'appel (1), une capacité en lecture sur le fichier (*capa_fichier*) à l'agent appelé. Cette capacité permet à l'objet *imprimante* de lire le contenu du fichier (2).

4.2 Echanges de Capacités

Comme on l'a vu dans la section précédente, l'emploi de capacités logicielles donne un modèle où les droits peuvent être échangés dynamiquement entre agents. Il reste alors à définir comment un programmeur d'agent va spécifier les règles d'échange de droits avec les autres agents.

Une des motivations fortes pour le modèle de protection que nous proposons est la modularité. En effet, nous n'avons pas voulu fournir des extensions au langage de programmation, permettant de spécifier le passage de capacités en paramètre lorsqu'une méthode est appelée sur un objet appartenant à un autre agent. Cela aurait alourdi et complexifié les programmes.

Notre proposition est d'exprimer les transferts de capacités entre agents à l'aide d'un langage de définition d'interface (IDL). Etant donné qu'une interface peut être décrite indépendamment du code qui la réalise, décrire la politique d'échange des droits au niveau de l'interface permet de séparer clairement l'expression de la protection du code de l'application.

Ainsi, un IDL a été étendu pour permettre de spécifier les capacités qui doivent accompagner le passage de référence en paramètre dans un appel de méthode. Cet IDL permet de définir des *vues.*

Une vue est une interface incluant la définition d'une politique de protection. Une vue est associée à une capacité et décrit :

- les méthodes qui sont autorisées par cette capacité,
- les échanges de capacités lorsqu'il y a passage de référence en paramètre dans une des méthodes de la vue. Les capacités à passer sont décrites en termes de vues.

La définition des vues est naturellement récursive, car une vue décrit les capacités qui accompagnent certains paramètres, cette description se faisant en terme de vues. Pour ce faire, un nom est associé à chaque vue lors de sa déclaration.

Dans l'exemple du serveur d'impression décrit auparavant, deux vues peuvent être définies pour la classe *File* : une vue *reader* qui n'autorise que la méthode *read* et une vue *writer* qui autorise les deux méthodes *read* et *write*. Pour la classe *Printer*, on définit la vue *user* qui autorise la méthode *print* dont la signature est la suivante : void *print* (***reader*** *File f*). Cette signature indique que lorsqu'une référence à un fichier est passée en paramètre de la méthode *print*, une capacité avec la vue *reader* doit être transmise à l'agent appelé.

Nous pourrions nous en tenir à une définition de la protection du coté de l'appelé si nous considérions une architecture client/serveur où la protection vise essentiellement à protéger le serveur contre ses clients. Cependant, dans le domaine des agents mobiles, nous devons prendre en compte la suspicion mutuelle entre les agents. Chaque agent doit pouvoir contrôler les capacités qu'il exporte aux autres agents (l'appelant comme l'appelé).

De plus, nous voulons assurer l'autonomie des agents. Plus précisément, il n'est pas possible pour un programmeur d'agent de vérifier la politique de protection d'un agent offrant un service (même si cette définition est enregistrée dans un serveur officiel) lors de la programmation de l'agent, car à cet instant, le programmeur ne sait pas encore forcément quels agents il va rencontrer.

Pour ces deux raisons, chaque agent va définir sa propre vision de la politique de protection à mettre en œuvre lorsqu'il interagit avec d'autres agents. A une capacité sont donc associées deux vues, la vue de l'agent appelant et la vue de l'agent appelé.

La vue de l'agent appelé décrit :

- Les méthodes autorisées.
- Pour chaque paramètre à l'aller (référence *R* reçue par l'appelé), la vue décrit les capacités qui seront données par l'agent appelé lorsque la référence *R* est utilisée pour un appel de méthode. Cette vue décrit donc, du point de vue de l'appelé, les capacités que l'agent accepte d'exporter.
- Pour chaque paramètre de retour (référence *R* donnée par l'appelé), la vue décrit la capacité retournée avec la référence *R*.

et la vue de l'agent appelant décrit :

- Pour chaque paramètre à l'aller (référence *R* donnée par l'appelant), la vue décrit la capacité donnée avec la référence *R*.

- Pour chaque paramètre de retour (référence *R* reçue par l'appelant), la vue décrit les capacités qui seront données par l'agent appelant lorsque la référence *R* est utilisée pour un appel de méthode. Cette vue décrit donc, du point de vue de l'appelant, les capacités que l'agent accepte d'exporter.

Ce schéma symétrique est la réponse aux problèmes de la suspicion mutuelle et de l'autonomie des agents. Chaque agent (l'appelant et l'appelé) définit les capacités qu'il accepte d'exporter.

Chaque agent définit les vues correspondant aux interfaces Java qu'il utilise. Ces vues sont prises en compte comme suit. Lorsqu'un agent exporte une référence à un objet en utilisant le serveur de nom décrit en section 3.2, il spécifie la vue associée à cette référence, donc la capacité qu'il désire exporter. De cette façon, l'agent définit également toutes les capacités qui seront exportées suite à des appels de méthode en utilisant la capacité exportée dans le serveur de nom (ces exportations sont spécifiées par la définition des vues de l'agent).

Lorsqu'un agent récupère une capacité dans le serveur de nom, il définit la vue qu'il désire associer à cette capacité (de son coté). De cette façon, il définit également toutes les capacités qui seront exportées suite à des appels de méthode en utilisant la capacité récupérée dans le serveur de nom.

Toute capacité échangée entre ces deux agents prendra en compte les vues des deux agents.

4.3 Exemple

Afin d'illustrer le schéma d'expression de notre modèle de protection, nous reprenons l'exemple du serveur d'impression (légèrement modifié) et donnons ci-dessous les interfaces Java mises en jeu et les vues qui sont définies par l'agent client et par l'agent serveur d'impression.

```
interface Printer_itf {
   void init ();                          // initialiser l'imprimante
   Job_itf run (Text_itf text);           // envoyer un texte à imprimer
}
interface Text_itf {
   String read();                         // lire le texte
   void write (String s);                 // écrire le texte
}
interface Job_itf {
   void stop ();                          // arrêter l'impression
}
```

Ces interfaces sont les interfaces Java qui sont mises en commun entre les agents pour pouvoir coopérer. Pour mettre son service à disposition des clients, le serveur d'impression exporte dans le serveur de nom une instance de sa classe *Printer*. Cette classe est une réalisation de l'interface *Printer_itf*. De son coté, l'agent client récupère cette instance et peut appeler une méthode (*init* ou *run*) sur cette instance grâce à l'interface *Printer_itf* (cette interface Java est utilisée pour forcer le type de l'instance reçue du serveur de nom).

Lorsqu'un fichier doit être imprimé, le client appelle la méthode *run* en passant une référence sur l'objet fichier à imprimer qui est de classe *Text*. Cette classe est une réalisation de l'interface *Text_itf*. La méthode *run* retourne une référence à une instance de la classe *Job* qui est une réalisation de l'interface *Job_itf*. Le client peut appeler la méthode *stop* sur cette instance pour arrêter l'impression s'il le désire.

La spécification de la protection pour ces deux agents vise à éviter les problèmes suivants :

- le serveur d'impression ne veut pas que le client puisse appeler la méthode *init* et réinitialiser l'imprimante,
- le client ne veut pas que l'imprimante puisse appeler *write* sur le texte et le modifier.

Dans notre schéma de protection, le client et le serveur définissent alors les vues suivantes :

client

```
view client implements Printer_itf {
    void init ();
    Job_itf run (Text_itf text pass reader);
}
view reader implements Text_itf {
    String read();
    void not write (String s);
}
```

serveur d'impression

```
view server implements Printer_itf {
    void not init ();
    Job_itf run (Text_itf text);
}
```

Chaque agent définit un ensemble de vues définissant sa politique de protection. Chaque vue « implemente » l'interface Java correspondant au type des objets qu'elle protège. Le mot clé **not** est placé devant une définition de méthode qui est interdite dans une vue. Lorsqu'une référence est passé en paramètre dans une vue, le programmeur peut spécifier la vue à passer avec le mot clé **pass**. Si aucune vue n'est spécifiée, cela signifie qu'aucune restriction n'est opéré pour ce passage de paramètre.

Dans cet exemple, le serveur d'impression définit la vue *server* qui interdit aux clients les appels à la méthode *init.* Aucune restriction n'est faite sur les paramètres à l'aller et au retour de la méthode *run*. Le client déclare la vue *client* qui spécifie que, lorsqu'une référence

sur un texte est passée à la méthode *run*, la vue *reader* doit être passée. La vue *reader* interdit la méthode *write* au serveur d'impression. Notons que le client n'a pas de raison de s'interdire la méthode *init* ; c'est une décision du serveur d'impression.

Lorsque le serveur d'impression enregistre une instance de la classe *Printer* dans le serveur de nom, il y associe sa vue *server*. Lorsque le client récupère cette référence du serveur de nom, il y associe sa vue *client*. Ces deux vues ainsi que celles qu'elles utilisent (*reader*) servent au contrôle des échanges des droits entre les agents.

En résumé, chaque agent spécifie sa politique de protection de façon autonome et modulaire, sous la forme de vues qui sont des interfaces incluant la définition des échanges de capacités entre les agents.

5 Expérimentations

Le modèle de protection décrit dans la section précédente à été réalisé sur Java. Dans cette section, nous décrivons cette réalisation, puis nous présentation comment nous avons validé ce modèle en prototypant une première application.

5.1 Réalisation sur Java

Pour l'implantation de ce modèle de protection, nous avons utilisé le fait que les références à des objets Java sont presque des capacités. En effet, étant donné que Java est un langage sûr, il n'est pas possible dans un programme Java de forger une référence à un objet et d'appeler une méthode sur cet objet. Ceci implique que si un objet O1 crée un objet O2, l'objet 02 n'est pas accessible à partir des autres objets de l'environnement Java, tant que O1 ne passe pas explicitement une référence vers O2 à ces autres objets. Ce passage ne peut se faire que par passage de paramètre lorsqu'un objet appelle O1 ou lorsque O1 appelle un autre objet. Ainsi, tant qu'un agent ne passe pas des références à ses objets, ceux-ci sont protégés contre les autres agents. Une référence Java peut donc être vue comme une capacité, mais ce n'est pas réellement une capacité, puisqu'il n'est pas possible de restreindre les méthodes pouvant être appelées sur cette référence. Nous avons donc ajouté un mécanisme inspiré de la notion de Proxy [Shapiro86] permettant de restreindre les droits associés à une référence Java.

Notre implantation repose sur la notion d'objets *filtres* que nous insérons entre l'appelant et l'appelé. Pour chaque vue définie par un agent, une classe filtre est générée et une instance de cette classe est insérée pour protéger l'agent.

Lorsqu'une référence d'objet est transmise à un autre agent (en paramètre d'un appel de méthode), nous passons à la place de la référence sur l'objet une référence sur une instance de la classe filtre, générée à partir de la vue spécifiée par l'agent qui donne la référence.

Cette classe filtre implante toutes les méthodes définies dans l'interface que la vue réalise. Elle déclare une variable d'instance qui pointe sur la référence de l'objet réellement

passé en paramètre et qui est utilisée pour retransmettre les appels de méthode autorisés vers le bon objet. Si une méthode non autorisée est appelée sur l'instance filtre, une exception est déclenchée.

Cette instance filtre, qui remplace la référence passée en paramètre d'un appel de méthode, est insérée par l'agent appelant ou plus précisément par l'instance filtre utilisée pour l'appel de méthode. Ce sont donc les objets filtres qui, lorsqu'ils sont appelés, sont chargés d'installer des objets filtres pour les références passées en paramètre.

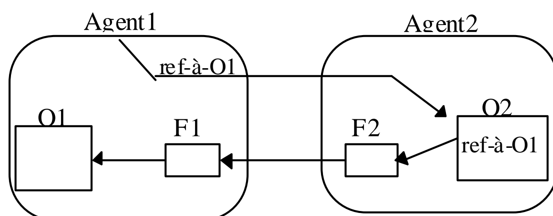

Figure 2. Gestion des objets filtres.

Réciproquement, lorsqu'une référence est reçue par un agent, une instance filtre est passée à la place, dont la classe a été générée à partir de la vue spécifiée par l'agent qui reçoit la référence.

Ainsi, deux objets filtres sont insérés entre l'appelant et l'appelé comme cela est représenté sur la figure 2.

Supposons qu'une référence à un objet *O1* est passée en paramètre d'un appel à un objet *O2* entre les agents *Agent1* et *Agent2*, que ce soit un paramètre à l'aller ou au retour. Les objets filtres installés entre *Agent1* (l'appelant) et *O2* (l'appelé) vont alors insérer les objets filtres suivants entre *Agent2* et *O1* :

- *F1* : l'objet filtre correspondant à la vue spécifiée par *Agent1* pour le paramètre *O1*. Ce filtre assure que seules des méthodes autorisées peuvent être appelées par *Agent2* et il insère des filtres pour le compte de *Agent1* pour les paramètres des appels à *O1*.
- *F2* : l'objet filtre correspondant à la vue spécifiée par *Agent2* pour le paramètre *O1*. Ce filtre insère des filtres pour le compte de *Agent2* pour les paramètres des appels à *O1*.

Voici ce que donne notre réalisation sur l'exemple du serveur d'impression. Les classes filtres générées sont données ci-dessous.

client		**serveur d'impression**
public class **reader** implements Text_itf {	|	public class **server** implements Printer_itf {

```
Text_itf obj;
public reader(Text_itf o) {
    obj = o;
}
public String read() {
    return obj.read();
}
public void write(String s) {
    Exception !!!
}
```

```
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        Exception !!!
    }
    public Job_itf run(Text_itf text) {
        return obj.run(text);
    }
}
```

```
public class client implements Printer_itf {
    Printer_itf obj;
    public Printer_stub(Printer_itf o) {
        obj = o;
    }
    public void init() {
        obj.init();
    }
    public Job_itf run(Text_itf text) {
        reader stub = new reader(text);
        return obj.run(stub);
    }
}
```

La classe *reader* du client retransmet les appels à *read* vers l'instance réelle, mais elle ne laisse pas passer les appels à *write.* De même, la classe *server* du serveur d'impression ne laisse passer que les appels à *run*. Dans la classe *client* du client, la méthode *run* prend en paramètre un objet *text* pour lequel une capacité avec la vue *reader* doit être passée. Pour ce paramètre, la méthode *run* de la classe filtre crée une instance de la classe filtre *reader* et l'initialise avec le vrai paramètre, puis retransmet l'appel en passant l'objet filtre créé à la place du vrai paramètre.

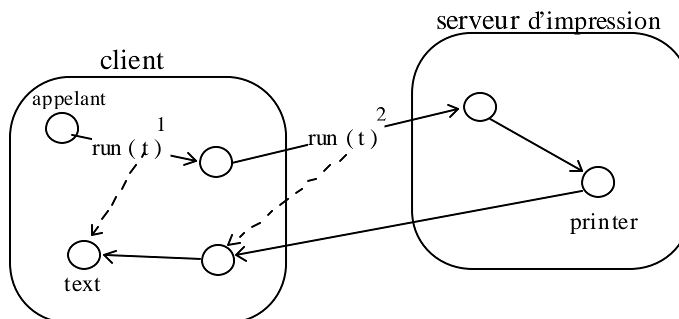

Figure 3. Objets filtre dans le serveur d'impression.

La figure 3 illustre l'insertion des objets filtres dans le cas du serveur d'impression. A l'étape (1), l'appel de la méthode *run* transmet la référence vers le vrai objet texte. Cet appel est réalisé sur l'objet filtre du client pour la référence à l'objet *printer*. Cet objet filtre crée un objet filtre pour le paramètre texte et transmet l'appel à l'objet filtre du serveur d'impression (2) en passant l'objet filtre du paramètre. L'appel est ensuite transmis au vrai objet *printer*. Ultérieurement, l'appel de la méthode *read* sur l'objet texte passe à travers l'objet filtre du texte installé par le client.

Notons que si le client avait défini une vue *vue_job* pour la référence à l'instance de la classe *Job* retournée par la méthode *run*, la classe filtre *client* aurait alors la méthode *run* suivante :

```
public Job_itf run(Text_itf text) {
   reader stub = new reader(text);
   return new vue_job(obj.run(stub));
}
```

Cette méthode crée à la fois un objet filtre pour le paramètre (*Text*) à l'aller et pour le paramètre (*Job*) au retour. Dans le cas du serveur d'impression, la vue *vue_job* n'est pas nécessaire.

5.2 Exemple d'Application

L'exemple du serveur d'impression que nous avons utilisé jusqu'ici ne visait qu'à expliquer le modèle et son implantation sur un exemple très simple.

Afin de valider le modèle de protection, nous avons développé une application dans laquelle des agents coopèrent tout en étant mutuellement méfiants; il s'agit d'un agenda coopératif. Dans le scénario d'exécution que nous avons implanté, chaque utilisateur garde une copie de son agenda sur son ordinateur personnel portable, mais il peut également envoyer un agent chargé d'organiser une réunion entre les membres d'un groupe. Cet agent contient une copie de l'agenda de l'usager et il va visiter les agendas des autres usagers du groupe pour chercher une date semblant convenir à tous le monde. Un des avantages de la programmation par agents est que l'usager peut envoyer son agent et déconnecter son portable. L'agent attendra de pouvoir contacter les agendas des différents membres du groupe, puis il attendra de pouvoir revenir sur la machine de l'usager (lorsque celle-ci sera reconnectée).

Pour ce qui est de la protection, si l'on veut permettre la coopération entre les différents agendas, on ne veut pas qu'un agenda puisse lire librement le contenu des agendas des autres usagers. Un agenda ne doit être autorisé qu'à chercher de la place libre dans les autres agendas.

Cette expérience nous a confirmé que la définition de la protection au niveau de l'interface était naturelle donc beaucoup plus facile. Après avoir mis au point cette application, la mise en place de la politique de protection a été presque immédiate.

6 Comparaison avec l'Existant

La recherche dans le domaine des agents mobiles est très active. Plusieurs projets de recherche développent des systèmes à agents mobiles et adressent le problème de la sécurité.

De façon générale, les principaux problèmes qui sont traités par tous ces systèmes sont les problèmes de l'isolation et de l'authentification. Pour ce qui est de l'isolation, tous reposent sur l'utilisation d'un langage sûr pour garantir qu'un agent ne peut pas manipuler des données privées de la machine ou d'autres agents. Dans ce contexte, notre approche a été de reposer sur un environnement incluant un langage sûr (en l'occurrence Java) et de nous attaquer aux lacunes des environnements fournis : l'expression de la politique de contrôle d'accès. Quand à l'authentification, il s'agit d'un aspect important qui a des répercussions sur le modèle de contrôle d'accès. Comme cela a été souligné en section 2, nous comptons utiliser un mécanisme d'authentification pour identifier des agents et pour permettre à un agent arrivant sur un site d'avoir un environnement de droits initial (un ensemble de capacités initial).

Concernant le contrôle de l'accès aux objets, nous pouvons comparer notre travail à trois projets en cours: AgentTcl [GCKR96], AgletIbm [IBM96] et Telescript [Magic96].

AgentTcl est un système à agents mobiles permettant d'écrire des programmes à l'aide du langage interprété Tcl. La protection dans AgentTcl [Gray96] repose sur l'utilisation d'une extension de Tcl appelée SafeTcl [OLW96], qui permet d'exécuter un programme sur un interprète (dit sûr) fournissant une restriction du jeu d'instruction de Tcl. Il est possible de redéfinir des opérations de l'interprète sûr avec un mécanisme d'alias, donc d'insérer des contrôles d'accès pour ces opérations. AgentTcl propose d'exécuter des agents différents sur des interprètes sûrs différents en fonction de l'agent, celui-ci étant authentifié par AgentTcl. Cependant, AgentTcl ne permet pas, à notre connaissance, le partage d'objets autres que des fichiers et il ne propose pas de modèle de protection permettant de spécifier une politique de gestion des droits d'accès.

Dans AgletIbm, le partage de données entre agents est seulement possible par partage de fichiers. AgletIbm est construit sur l'environnement Java et permet au niveau de chaque site de déterminer les fichiers qui sont accessibles aux agents visitant le site. AgletIbm fournit un mécanisme permettant d'authentifier les agents. Cependant, il faut spécifier les fichiers accessibles aux agents « à la main ». Cette spécification est réalisée par l'administrateur du site et aucun modèle n'est fourni pour permettre de construire des politiques de gestion des droits d'accès au niveau des applications.

Dans Telescript, les agents peuvent interagir par partage d'objet. Telescript offre des mécanismes permettant d'authentifier les agents auxquels sont associées des attributs, notamment un identifiant unique d'agent et la machine d'où ils proviennent. Le contrôle d'accès doit être réalisé dans les programmes des agents qui doivent eux-mêmes tester si l'opération est autorisée en fonctions des attributs de l'agent appelant. Par rapport à Telescript, notre proposition diffère par le fait que nous séparons la définition de la protection du code de l'agent, la protection étant définie au niveau de l'interface de l'agent. Cette modularité rend plus facile la définition de la protection. De plus, notre modèle est basé sur des capacités logicielles, ce qui permet aux agents d'échanger des droits dynamiquement. Dans Telescript, fournir cette dynamicité dans une application reste possible, mais complexe à réaliser.

7 Conclusion et Perspectives

Dans cet article, nous avons décrit un modèle de protection permettant de définir les règles de contrôle d'accès d'un agent mobile.

La définition de ces règles est réalisée au niveau de l'interface de l'agent, ce qui la facilite et la rend plus claire. Le modèle est basé sur des capacités logicielles, permettant ainsi l'échange de droits dynamiquement entre des agents mutuellement méfiants. Chaque agent définit sa propre politique de protection et les politiques spécifiées par chaque agent interagissant sont prises en compte à l'exécution.

Ce modèle de protection a été réalisé sur l'environnement Java et les expérimentations avec des applications simples ont montré l'intérêt de cette approche.

Ce travail se poursuit actuellement. L'objectif principal est de réaliser un système à agents mobiles plus complet permettant d'expérimenter avec des applications plus importantes. Nous comptons ainsi identifier plus précisément les besoins en termes de contrôle d'accès de ces applications et compléter notre proposition, notamment avec des mécanismes d'authentification permettant d'associer un environnement de droit initial à chaque agent.

Bibliographie

[GCKR96] R. Gray, G. Cybenko, D. Kotz, and D. Rus: "Agent TCL", Department of Computer Science, Dartmouth College, Hanover, NH, May 29, 1996. URL: *http://www.cs.dartmouth.edu/?agent/papers/chapter.ps.Z*.

[GM95] J. Gosling and H. McGilton, The Java Language Environment: a White Paper, Sun Microsystems Inc., 1995http://java.sun.com/whitePaper/java-whitepaper-1.html,

[Gray96] R. Gray : "A Flexible and Secure Mobile-Agent System," Fourth Annual TC/TK Workshop '96 - July 10-13, 1996.

[IBM96] Iternational Business Machines Corporation: "Mobile Agent Facility Specification," OMG TC Document cf/96-08-01, available on *http:/www.trl.ibm.co.jp/aglets/maf.ps*, August 24, 1996.

[KGR96] D. Kotz, R. Gray, and D. Rus : "Transportable Agents Support Worldwide Applications", Department of Computer Science, Dartmouth College, Hanover, NH, March 1, 1996.

[Levy84] H. M. Levy. *Capability-Based Computer Systems*, Digital Press, 1984.

[Magic96] Magic Inc. "An Introduction to Safety and Security in Telescript", Magic Inc. document, available on http://www.genmagic.com/Telescript/security.html

[OLW96] J. Ousterhout, J. Levy, andB. Welch : "The Safe-TCL Security Model", Draft, Novembre 16, 1996.

[Shapiro 86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle, *Proc. of the 6th International Conference on Distributed Computing Systems*, pp. 198-204, 1986.

[Sun96] Sun Microsystems, "JDK 1.1 Documentation", Sun Microsystems, available on http://www.javasoft.com/products/jdk/1.1/docs/index.html