

Rapport de DEA Réseaux et Systèmes Distribués  
Arnaud Contes

# Programmation à objets distribués : Sécurisation de collecticiels

INRIA Sophia-Antipolis, Equipe OASIS

Mars-Juin 2001

## Remerciements

Je remercie en premier Denis Caromel qui m'a encadré tout au long de ce stage.

Je remercie Fabrice et Julien pour leurs nombreux conseils, leur patience, leur amitié.

Je remercie aussi tous les membres de l'équipe OASIS.

Je remercie particulièrement Laurent, qui a été un joyeux compagnon pendant ces quatre mois de stage. :)

## Table des matières

<b>Remerciements</b>	<b>2</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Objectifs . . . . .	5
1.2 Sécurité informatique . . . . .	5
1.3 Définitions . . . . .	6
1.4 Problèmes spécifiques de la sécurité distribuée . . . . .	6
1.5 Introduction à la cryptographie . . . . .	8
<b>2 Contexte et état de l'art</b>	<b>10</b>
2.1 Présentation de ProActive PDC . . . . .	10
2.2 Etat de l'art des modèles de sécurité existants . . . . .	13
2.2.1 Les Méta Objets Sécurisés . . . . .	13
2.2.2 Jini . . . . .	17
2.2.3 Ajanta . . . . .	18
2.2.4 Modèle de sécurité . . . . .	19
2.3 Outils existants . . . . .	20
2.3.1 Remote Methode Invocation et sérialisation . . . . .	20
2.3.2 Java Authentication and Authorization Service (JAAS) . . . . .	20
2.3.3 Java Secure Socket Extension (JSSE) et Java Cryptography Extension (JCE) . . . . .	21
2.3.4 Remote Methode Invocation Secure Extension (RMISE) . . . . .	21
<b>3 Un modèle de sécurité pour ProActive</b>	<b>22</b>
3.1 Principes . . . . .	22
3.2 Les Domaines . . . . .	22
3.3 Politiques de sécurité . . . . .	23
3.4 Modes de communications . . . . .	24
3.5 Infrastructure à clé publique . . . . .	26
<b>4 Implémentation avec des Protocoles à Méta-Objets (MOP)</b>	<b>28</b>
4.1 Création des objets actifs sécurisés . . . . .	28
4.2 Création des domaines . . . . .	28
4.3 Sécurisation des appels . . . . .	30
4.4 Benchmarks . . . . .	31
<b>5 Conclusion</b>	<b>33</b>
<b>A Grammaire de la politique de sécurité</b>	<b>34</b>

## Table des figures

1.1	Chiffrement et déchiffrement avec une clé (Algorithme symétrique) . . . . .	8
1.2	Chiffrement et déchiffrement avec deux clés (Algorithme asymétrique) . . . . .	8
2.1	Graphe avec des objets actifs . . . . .	10
2.2	Schéma d'un objet actif . . . . .	11
2.3	Invocation de méthode sur un objet actif distant . . . . .	12
2.4	Représentation d'une JVM avec plusieurs nodes . . . . .	12
2.5	Forwarders . . . . .	13
2.6	Une référence sur un objet avec un SMO . . . . .	13
2.7	Sortie du domaine virtuel d'un SMOp . . . . .	15
2.8	Suppression automatique des SMOs à la sortie d'un domaine . . . . .	15
2.9	Les références sans SMO pointant vers l'intérieur d'un domaine sont dangereuses . . . . .	16
2.10	Les références sans SMO pointant vers l'intérieur d'un domaine sont dangereuses . . . . .	16
2.11	Invocation de méthode sur un autre domaine . . . . .	16
3.1	Principe de certification . . . . .	26
4.1	Schéma d'un objet actif gérant la sécurité . . . . .	30
4.2	Surcoût du chiffrement, temps en ms . . . . .	32

---

## Chapitre 1

# Introduction

### 1.1 Objectifs

Mon stage s'est déroulé à l'INRIA de Sophia-Antipolis dans l'équipe OASIS.

Les objectifs du stage sont:

- Etudier les politiques de sécurité spécifiques aux applications distribuées.
- Définir et implémenter des primitives de sécurité au sein de ProActive.

Ce document est structuré en cinq parties. Nous allons tout d'abord procéder à un tour d'horizon rapide sur la sécurité informatique, introduire les notions essentielles en sécurité.

Le chapitre 2 présente la librairie ProActive et un état de l'art de la sécurité des applications réparties.

Dans le chapitre 3, nous présentons formellement le modèle de sécurité que nous avons élaboré pour la librairie ProActive.

Le chapitre 4 présente l'implémentation du modèle de sécurité qui a été présenté dans le chapitre précédent.

Enfin, dans le chapitre 5, nous concluons et présentons les ouvertures possibles.

### 1.2 Sécurité informatique

La sécurité d'un système informatique a pour but de protéger les informations sensibles contre une utilisation non autorisée (accidentelle ou mal-intentionnée).

L'objectif essentiel est d'assurer les deux propriétés suivantes :

- la *confidentialité* qui garantit que les informations du système ne sont ni rendues accessibles, ni divulguées à une entité non autorisée.
- l'*intégrité* qui garantit que les informations du système ne sont pas altérées ou détruites par une entité non-autorisée.

La *politique de sécurité* d'un système informatique permet de spécifier les actions autorisées dans ce système, c'est-à-dire les actions qui n'invalident pas les propriétés précédentes.

Pour contrôler l'accès aux informations sensibles d'un système, une *politique de sécurité* doit identifier les objets du système contenant les informations sensibles, les opérations permettant d'accéder à ces informations, ainsi que les sujets manipulant ces informations.

Ainsi, une politique de sécurité repose sur :

- la définition des ensembles de ses sujets, de ses objets, et des opérations permettant d'accéder aux objets.
- la définition de l'ensemble des règles pour le contrôle d'accès entre les sujets et les objets.

Nous pouvons distinguer deux types de sécurité, la *sécurité centralisée* et la *sécurité distribuée*. Dans le système centralisé, il existe une unique politique de sécurité prenant en compte l'ensemble des entités du système. A l'inverse la sécurité distribuée se caractérise par la coexistence de nombreuses politiques de sécurité [GGKL89].

### 1.3 Définitions

**Principal ou sujet :** l'entité dans un système informatique à laquelle on peut donner des autorisations. C'est l'unité de base dans le système d'authentification.

**Access Control List ou Liste de Contrôle d'Accès (ACL) :** liste de sujets autorisés à accéder à une ressource

**Authentifier :** vérifier l'identité d'une personne ou d'un agent externe au système d'authentification.

**Capability :** représentation de la preuve incontestable que la personne qui le présente est autorisée à accéder à la ressource nommée dans le ticket.

### 1.4 Problèmes spécifiques de la sécurité distribuée

L'un des problèmes les plus importants en sécurité distribuée concerne l'interaction de différents systèmes, chaque système pouvant avoir ses propres contraintes de sécurité.

#### Composition de politiques de sécurité

La sécurité distribuée se caractérise par l'existence de plusieurs politiques de sécurité qui peuvent donner une politique de sécurité globale incohérente.

A partir de ces politiques de sécurité différentes, on peut se ramener soit à la *cohabitation* des politiques, soit à leur *fédération*.

#### Cohabitation des politiques de sécurité

Deux politiques de sécurité *cohabitent* lorsqu'elles ne sont pas définies pour les mêmes entités du système et qu'aucun sujet de l'un n'accède aux objets de l'autre.

Il n'est pas nécessaire de détailler davantage la cohabitation des politiques de sécurité, le système résultant pouvant être assimilé à deux systèmes indépendants, chacun mettant en œuvre une politique de sécurité centralisée.

#### Fédération de politiques de sécurité

La fédération de deux politiques se pose lorsqu'il existe des interactions entre les entités relevant de deux politiques de sécurité différentes. L'approche pour supporter la fédération de politiques de sécurité s'appuie sur la composition des politiques, c'est-à-dire la composition de leurs règles respectives. Notons que la composition de deux politiques de sécurité repose sur une *politique de composition* définissant les *règles de la composition* telles que les accès autorisés entre les sujets et les objets des différentes politiques de sécurité. Nous identifions deux approches à la composition de deux politiques de sécurité :

- l'*interopération* de politiques de sécurité qui garantit que la politique *résultante* préserve les conditions de sécurité associées à chaque politique *initiale*.
- la *combinaison* de politiques de sécurité qui spécifie la politique *résultante* à partir des spécifications des politiques *initiales*.

Nous précisons ces deux approches dans les paragraphes qui suivent.

#### Interopération de politiques de sécurité

L'*interopération* de politiques de sécurité permet de composer des politiques de sécurité en préservant les règles définies par chacune d'elles. L'interopération est ainsi adaptée à la composition de politiques de sécurité décrites pour des systèmes mutuellement suspicieux. L'interopération permet à des systèmes distribués dont la sécurité est centralisée d'interagir dans un environnement distribué sans réduire le niveau de sécurité de chacun des systèmes.

Un *modèle d'interopération* obtenu en étendant la notion de modèle de sécurité permet de vérifier que le système résultant préserve bien les règles initiales de chacun des systèmes de sécurité.

Plusieurs modèles d'interopérations ont été présentés dans la littérature. Parmi les plus cités nous retrouvons le modèle d'interopération de Gong et Qian [GQ96], celui de McCullough [Mtf90] ou encore celui de McLean [McL88].

### Combinaison de politiques de sécurité

La *combinaison* de politiques de sécurité permet de spécifier la politique *résultante* à partir des spécifications des politiques *initiales*. La politique résultante peut ne pas garantir les conditions de sécurité associées à chaque politique initiale, voire invalider certaines de leurs règles.

La combinaison de politiques de sécurité permet à deux systèmes de collaborer. A l'inverse de l'interopération de politiques de sécurité où les systèmes sont mutuellement suspicieux, la combinaison de politiques de sécurité peut être utilisée lorsqu'il existe un certain *degré de confiance* entre les différents systèmes. Cette collaboration se traduit par éventuellement la «relaxation» de certaines règles afin de combiner les deux politiques de sécurité. La politique résultante peut être perçue comme une extension ou une restriction des deux politiques de sécurité initiales.

Il existe plusieurs modèles de combinaisons dans la littérature.

Le modèle de combinaison basé sur une *algèbre de sécurité* étend le modèle de sécurité afin de pouvoir exprimer formellement diverses conditions de sécurité et introduit des opérateurs permettant de définir une *algèbre de sécurité*. McLean dans [McL96] propose une *algèbre booléenne* pour les politiques de contrôle d'accès multi-niveaux avec gestion dynamique des niveaux de sécurité, et définies sur un même système.

Foley dans [Fol92] propose un modèle de combinaison s'appuyant sur le langage de spécification Z. Le langage Z est basé sur la théorie des ensembles et la logique du premier ordre (cf. [Dil94]).

Hosmer [Hos92] adopte une approche ne reposant sur aucune relation d'ordre. Il introduit la notion de *métapolitique* ou «politique de politiques», une approche informelle pour décrire, faire collaborer, ou combiner des politiques de sécurité.

### Conclusion sur la sécurité distribuée

Le problème lié à l'existence de différentes politiques de sécurité est résolu en composant ces politiques à l'aide d'une *politique de composition*. Nous avons identifié deux approches à la composition de politique de sécurité. La première, l'interopérabilité de politiques de sécurité est recommandée dès lors que les systèmes informatiques sont mutuellement suspicieux. En particulier, l'interopération de politiques de sécurité ne peut se faire que lorsque les politiques de sécurité sont *cohérentes*. A l'inverse la seconde approche, la *combinaison* de politiques de sécurité, permet de composer des politiques de sécurité éventuellement *incohérentes*. Dans ce cas, la politique résultante est construite à partir des spécifications des politiques initiales.

La différence fondamentale entre les deux approches repose sur la mise en œuvre de la politique résultante. Lors de l'interopérabilité de deux politiques de sécurité, les mécanismes de protection de ces politiques sont utilisés afin de garantir la politique résultante. Lors de la combinaison de deux politiques de sécurité, les mécanismes mis en œuvre pour la politique résultante peuvent être en partie différents de ceux des politiques initiales. Il faut toutefois remarquer que la politique résultant de la combinaison ou celle résultant de l'interopération de deux politiques de sécurité peuvent être identiques.

## 1.5 Introduction à la cryptographie

### Algorithme cryptographique

Un algorithme cryptographique est une fonction mathématique utilisée pour le chiffrement<sup>1</sup> et le déchiffrement<sup>2</sup>. La cryptographie moderne utilise des algorithmes à clés. Une clé n'est rien d'autre qu'une valeur de taille fixée parmi un très grand nombre de valeurs possibles. Les opérations de chiffrement et de déchiffrement dépendent de ces clés. Avec ces algorithmes, toute la sécurité réside dans la (ou les) clé(s), et non dans le détail de l'algorithme. Ceci implique que l'algorithme peut être publié et analysé. En cryptographie, il y a deux grands types d'algorithmes :

**Les algorithmes à clés secrètes** (également appelés algorithmes symétriques) sont des algorithmes où la clé de chiffrement peut être calculée à partir de la clé de déchiffrement ou vice versa. Dans la plupart des cas, la clé de chiffrement et la clé de déchiffrement sont identiques. Pour de tels algorithmes, l'émetteur et le destinataire doivent se mettre d'accord sur une clé à utiliser avant d'échanger des messages. Cette clé doit être gardée secrète. La sécurité d'un algorithme à clé secrète repose sur la clé : si elle est dévoilée, alors n'importe qui peut chiffrer ou déchiffrer des messages avec elle. Le schéma 1.1, page 8 illustre de principe de chiffrement à base d'algorithme à clé secrète. D'une manière générale, les algorithmes à clés secrètes sont très rapides car ils consistent tout simplement à réarranger les bits d'un messages.

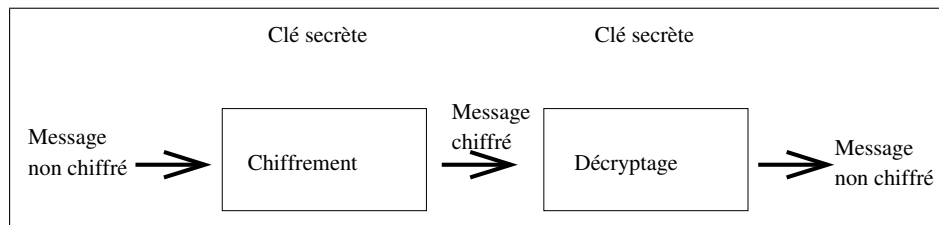


FIG. 1.1 – Chiffrement et déchiffrement avec une clé (Algorithme symétrique)

**Algorithme à clés publiques** : Les algorithmes à clés publiques (également appelés algorithmes asymétriques) sont différents. Ils sont conçus de telle manière à ce que la clé de chiffrement soit différente de la clé de déchiffrement. De plus, la clé de déchiffrement ne peut pas être calculée (du moins en un temps raisonnable) à partir de la clé de chiffrement. De tels algorithmes sont dits «à clé publique» parce que la clé de chiffrement peut être rendue publique : n'importe qui peut utiliser la clé de chiffrement pour chiffrer un message mais seul celui qui possède la clé de déchiffrement peut déchiffrer le message chiffré résultant. Dans de tels systèmes, la clé de chiffrement est appelée clé publique et la clé de déchiffrement est appelée clé privée. Le schéma 1.2, page 8 illustre de principe de chiffrement à base d'algorithme à clé publique. Les algorithmes à clés publiques reposent tous sur des problèmes mathématiques non-résolus à l'heure actuelle (factorisation de nombre premiers, problème du logarithme discret, . . .) . Par conséquent, ils sont beaucoup plus lent que les algorithmes symétriques.

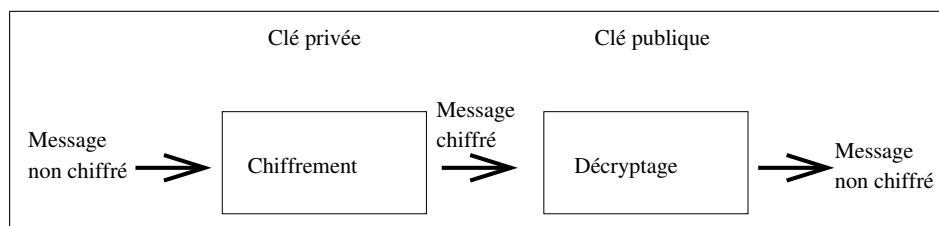


FIG. 1.2 – Chiffrement et déchiffrement avec deux clés (Algorithme asymétrique)

1. cryptage  
2. décryptage

## Protocole hybride

Dans certain cas, il peut-être judicieux de recourir à des techniques de chiffrement utilisant à la fois des techniques de cryptographie asymétrique et symétrique. L'un des cas les plus courant consiste à échanger une clé de session symétrique à l'aide de la cryptographie asymétrique. Ainsi seul l'établissement de la clé de session sera coûteux en temps.

## Signature numérique

Un des plus gros apport de la cryptographie à clés publiques est celui des méthodes de signature numérique. Les signatures numériques permettent au destinataire d'un message de vérifier l'authenticité de l'origine de ce message, et également de vérifier l'intégrité du message, c'est à dire de s'assurer qu'il n'a pas été modifié. Les signatures numériques permettent également d'assurer la non-répudiation d'un message, c'est à dire de faire en sorte que l'émetteur d'un message ne puisse pas nier l'avoir émis. Les fonctions d'intégrité, d'authenticité et de non-répudiation sont fondamentales en cryptographie. La signature numérique est nettement supérieure à la signature manuelle, car s'il est facile d'imiter une signature manuelle, il est en revanche presque impossible de contrefaire une signature numérique.

L'émetteur du message utilise sa clé privée pour chiffrer le message (et non pour le déchiffrer). Le message chiffré obtenu fait alors office de signature. L'opération de vérification de signature consiste à utiliser la clé publique de l'émetteur pour déchiffrer la signature (le message chiffré) et de comparer le résultat au message original en clair. Si les deux messages sont identiques, alors le destinataire est assuré de l'intégrité et de l'authenticité de message. De plus, l'émetteur ne peut pas nier avoir signé le message car il est le seul à posséder sa clé de signature (il s'agit d'une clé privée).

## Niveau de sécurité des algorithmes actuels

Le niveau de sécurité des algorithmes symétriques et asymétriques dépend en général directement de la taille des clés utilisées. L'évaluation de la solidité des algorithmes asymétriques est cependant assez délicate, car étant donné qu'ils reposent sur des problèmes mathématiques non résolus à l'heure actuelle, des découvertes majeures dans la Recherche des Mathématiques pourraient alors rendre ces algorithmes obsolètes.

Dans [LV00], les auteurs étudient le niveau de sécurité des algorithmes actuels, et estiment la taille des clés à utiliser pour garantir un niveau de sécurité optimal.

## Attaques classiques

**L'attaque exhaustive** : Cette attaque, très adaptée en particulier pour les crypto-systèmes symétriques, consiste à essayer une à une les différentes clés. Pour la plupart des algorithmes symétriques reconnus (DES, AES, ...), cette attaque reste la plus rapide. Néanmoins si la taille des clés est suffisante (supérieure à 128 bits), elle est irréalisable en un temps réduit.

**L'attaque par observation du texte chiffré** : Cette attaque consiste à déduire la clé en observant le texte chiffré. Elle consiste souvent à effectuer des tests statistiques sur la répartition des différents caractères.

**L'attaque à texte clair connu** : Cette attaque consiste à déduire la clé à partir d'un texte clair et du texte chiffré associé.

**L'impersonation (man in the middle attack)** est sans doute l'attaque la plus redoutée des concepteurs de crypto-systèmes. Cette attaque consiste à s'interposer entre deux partis désirant communiquer en toute confidentialité et à se faire passer pour l'un auprès de l'autre, et vice-versa. L'idée est donc d'intercepter le secret partagé par les deux partis nécessaire pour communiquer en toute confidentialité. Cette attaque s'applique aussi bien aux crypto-systèmes symétriques qu'aux crypto-systèmes asymétriques.

## Chapitre 2

# Contexte et état de l'art

## 2.1 Présentation de ProActive PDC

ProActive est une bibliothèque Java, conçue pour la programmation parallèle, distribuée et concurrente, ainsi que pour la métaprogrammation.

On se place dans le contexte d'un modèle MIMD (Multiples Instructions, Multiples Données) à objets actifs. A partir d'un ensemble restreint de primitives, une bibliothèque simple et flexible est définie pour le parallélisme, la distribution et la programmation concurrente.

ProActive n'ajoute aucune extension syntaxique à Java. Les programmeurs écrivent du code standard. La bibliothèque est elle-même extensible par les programmeurs, elle est un système ouvert aux optimisations et modifications.

La librairie ProActive est uniquement constituée de classes Java standards, et ne requiert aucune modification de la Machine Virtuelle Java, ni du compilateur, ni de rajout de preprocessing. ProActive PDC est basée sur la bibliothèque Java RMI standard.

ProActive fournit:

- des Objets Actifs
- des appels de méthodes asynchrones avec futurs transparents
- une communication par messages (Request and Reply)
- des agents mobiles (migration faible)

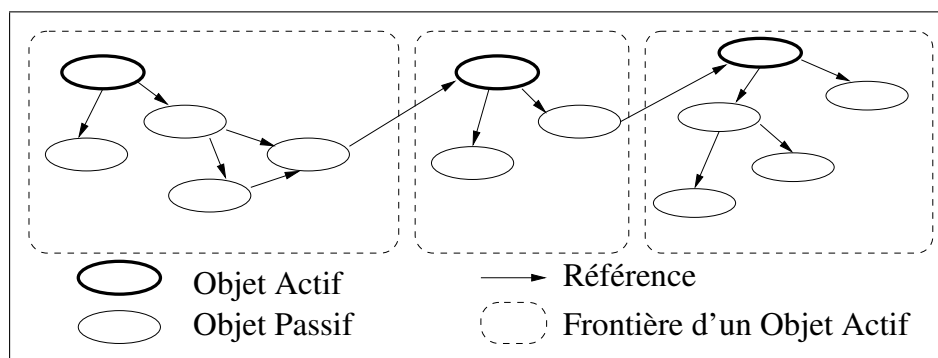


FIG. 2.1 – Graphe avec des objets actifs

La sémantique de ProActive est séquentielle, multi-threadée, et distribuée. Le polymorphisme entre objets standards et objets actifs permet une meilleure réutilisation du code. Le mécanisme de synchronisation automatique par futur (avec attente par nécessité) facilite la conception d'application collaborative. ProActive est interfacée avec RMI, bien sûr, mais aussi avec Jini [Wal98] et Globus [FK97].

### Object Actif

Un objet actif est un objet mono-threadé, il est construit explicitement à l'exécution du programme.

On lui ajoute deux méta-objets qui permettent la réification des méthodes et leur stockage pour une utilisation ultérieure. Ils fournissent de même un mécanisme de communication et des appels asynchrones.

ProActive permet ainsi d'évoluer dans un contexte hétérogène, on peut créer à la fois des objets actifs et des objets passifs.

Le premier méta-objet associé avec un objet actif s'appelle un *Body* et représente son activité (il possède la thread). Le second, le *Proxy* est un wrapper utilisé par les autres objets pour communiquer avec l'objet actif.

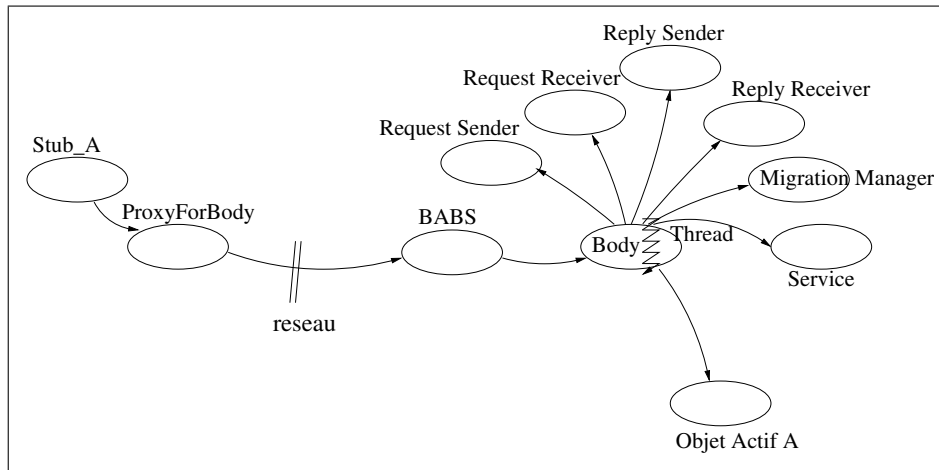


FIG. 2.2 – Schéma d'un objet actif

Si on considère la relation appelé-appelant, nous avons la relation suivante : chaque objet actif a un objet *Body*, et tous les appelants utilisent un objet *Proxy*.

### Protocole à Méta Objets

La librairie *ProActive* utilise les principes des Protocoles à Méta-Objets (MOP) [KdR91] dans le but de réifier deux des éléments constituant l'exécution d'un programme Java : les invocations de méthodes sur des objets et les appels aux constructeurs.

Selon la classification utilisée dans [CHV00], le MOP de ProActive est un *Proxy-based MOP*. Ce type de MOP introduit des modifications dans le programme afin de réifier des événements survenant pendant l'exécution tels que les invocations de méthodes ou les accès aux variables des classes. Les modifications dans le programme sont introduites à l'exécution en utilisant des objets qui implémentent le modèle *proxy* appelés aussi *wrapper*. Ce type de MOP ne requiert aucune modification de la Machine Virtuelle Java (JVM). Le code source de l'application n'est pas modifié.

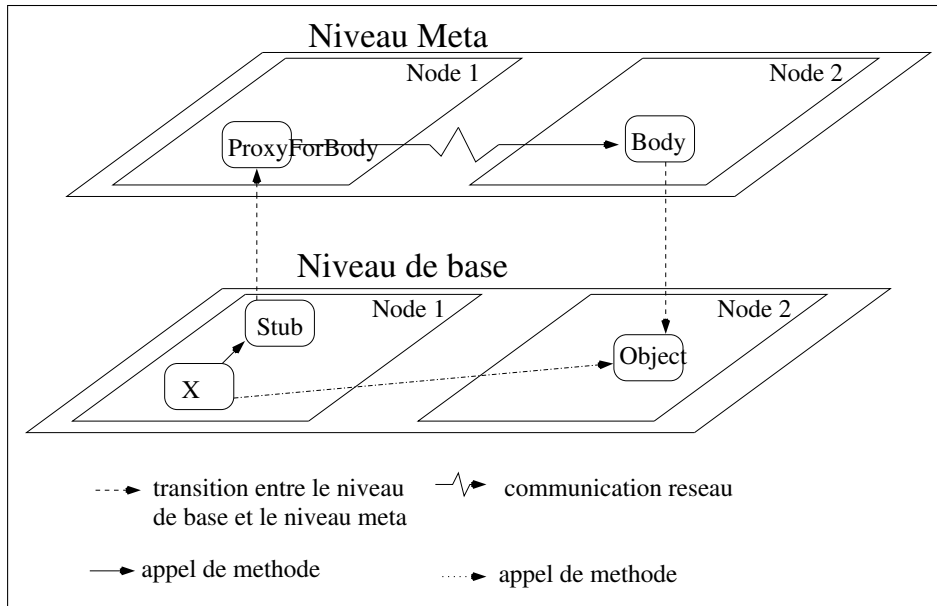
La figure 2.3 à la page 12 nous montre une invocation de méthode sur un objet actif distant. L'objet X croit appeler la méthode directement sur l'objet Object. En fait, il fait son appel sur l'objet *Stub* qui réifie l'appel de méthode, il le transforme en *MethodCall* et le passe au *ProxyForBody*. c'est l'objet *ProxyForBody* qui décide de la politique des messages (asynchrone, synchrone, sans retour). Il construit une requête avec les paramètres du *MethodCall* et l'envoie à l'objet *Body* correspondant. Le *Body* fait l'appel de méthode sur l'objet. Cet exemple n'est valable que pour une méthode sans paramètre de retour.

### Mobilité

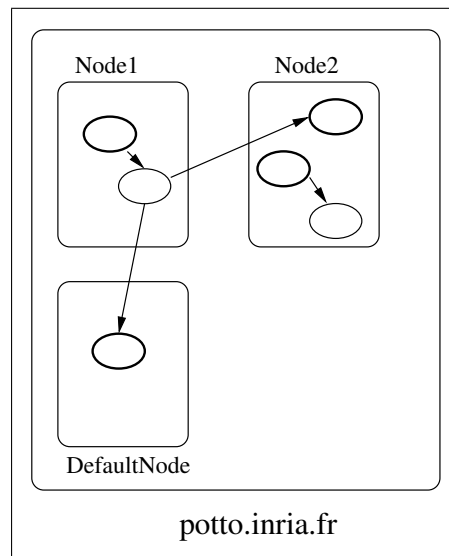
ProActive offre une migration faible des objets actifs. Un objet actif possède une file d'attente des requêtes. Lorsqu'un objet va vouloir migrer, il va terminer la requête en cours s'il y en avait une, arrêter de servir les requêtes de sa file d'attente. A ce moment, on peut sérialiser les données de l'objet actif et les faire migrer sans perte d'informations.

### Les Nodes

Les objets actifs ont la possibilité de migrer. Mais il faut pouvoir leur indiquer un endroit où aller. C'est à cette fin que les Nodes ont été créés, ils sont des points d'entrées dans les Machines Virtuelles

FIG. 2.3 – *Invocation de méthode sur un objet actif distant*

Java distantes ou locales. Chaque Node peut accueillir un ou plusieurs objets actifs et les objets passifs référencés par l'objet actif. Il n'y a pas dans les Nodes de notion de hiérarchie. Un Node appartient à une JVM qui elle-même appartient à un hôte généralement une machine avec une adresse IP. On peut avoir plusieurs JVM par machine et plusieurs Node par JVM.

FIG. 2.4 – *Représentation d'une JVM avec plusieurs nodes*

## Les Forwarders

Quand un objet actif migre, il laisse derrière lui un objet spécial appelé un *forwarder* dont la tâche est de faire suivre les appels jusqu'au nouvel emplacement de l'objet actif.

Le mécanisme de *ProActive* pour localiser l'objet distant est basé sur une simple remarque : un objet n'a besoin pas de connaître précisément où se trouve l'objet sur lequel il veut faire une requête mais il veut être sûr que chacun de ses messages va être délivré correctement.

Un objet actif doit donc pouvoir toujours être atteint. Pour cela, une fois un forwarder créé, il restera tant que l'objet actif vers lequel il redirige les requêtes n'a pas disparu. L'inconvénient majeur des

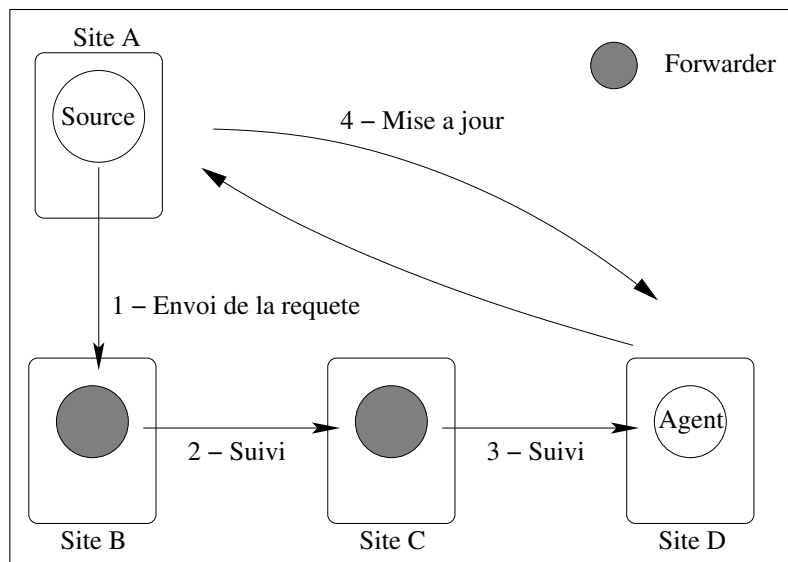


FIG. 2.5 – Forwarders

forwarders est l'augmentation de leur nombre. Plus un ordinateur hôte va abriter d'objets actifs, plus le nombre de forwarder qu'il devra continuer à héberger va augmenter.

## 2.2 Etat de l'art des modèles de sécurité existants

### 2.2.1 Les Méta Objets Sécurisés

#### Présentation

Dans la plupart des modèles basés sur des objets, on considère que les références sur les objets comme des *capabilities*<sup>1</sup>. Si un client a une référence sur un objet, il peut alors accéder à cet objet. Réciproquement, si un client n'a pas de référence sur cet objet, il n'a pas le droit d'accéder à un objet. Dans [RK98] les auteurs étendent simplement ce modèle en rajoutant la possibilité d'attacher un ou plusieurs objets spéciaux à une référence sur un objet. Ces objets spéciaux vont être invoqués pour chaque opération sur la référence de l'objet nécessitant des fonctionnalités liées à la sécurité. Ces objets ne sont pas visibles par l'application, les références sur des objets protégés ou sur des objets non protégés sont semblables. Ces objets sont considérés comme des *méta objets*. Ils sont appelés *Security Meta Object* (SMO).

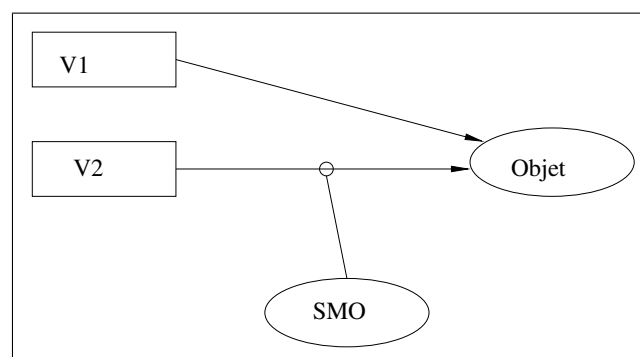


FIG. 2.6 – Une référence sur un objet avec un SMO

On peut avoir plusieurs références sur un objet, et sur chacune des ces références avoir un SMO différent. Tout le monde est autorisé à attacher un SMO à une référence. Si plusieurs SMO sont attachés

1. je garde le mot anglais

à une référence, ils sont appelés séquentiellement avant d'autoriser l'accès.

Il est possible qu'un même SMO soit attaché à plusieurs références. Il n'est pas possible d'enlever un SMO d'une référence à moins que ce dernier décide de s'en détacher lui-même. L'introduction de *capabilities* introduit trois concepts supplémentaires sur les références :

- la restriction des droits d'accès,
- la révocation,
- l'expiration.

La restriction d'accès est facilement implémentable en utilisant les informations passées au SMO à chaque appel. La révocation d'une *capabilities* peut être implémentée en attachant un SMO qui autorise par défaut l'accès à l'objet et qui désactive l'accès dès qu'une méthode de révocation est appelée sur cet SMO. Cela implique de garder une référence sur le SMO quand on le crée.

L'expiration est implémentée de façon similaire. Le SMO vérifie à chaque requête la date courante et la date d'expiration. Si la *capabilities* a expiré, l'accès est refusé.

Tous ces concepts peuvent être implémenté par un SMO spécial attaché à une référence ou par trois SMO différents attachés à une même référence.

Dans les systèmes qui n'utilisent que des *capabilities* se posent les problèmes de transitivité des droits d'accès.

### La transitivité des droits

On va pouvoir protéger une liste avec un SMO. En faisant cela, on voudrait pouvoir protéger tous les objets de la liste. Seulement pour le moment si l'accès à la liste est bien protégé, mais on ne peut empêcher la partie qui n'est pas de confiance d'obtenir des références sur les objets de la liste si elles sont passées comme résultat d'un appel de méthode sur les objets de la liste.

Par exemple, la liste peut avoir une méthode *Get* qui permet d'obtenir des objets de la liste. Les références sur ces objets ne sont pas protégées. Les références sur ces objets sont par défaut non protégées (à moins que la liste n'implémente sa propre politique de sécurité ce qui n'est pas prévu dans notre cas).

Pour réaliser cette transitivité sur le contrôle d'accès nous avons besoin de protéger toutes les références retournées comme résultat d'un appel de méthode. On peut faire cela facilement avec les SMO. Quand une référence est retournée par le biais d'un appel de méthode sécurisé, une méthode spéciale du SMO est appelée. On peut faire en sorte que le SMO s'attache lui-même et propage de cette façon sa politique de sécurité.

Ainsi aucun objet d'une partie qui n'est pas de confiance ne pourra accéder à un objet de la liste.

Le rôle des SMO est de fournir les informations des *principals*<sup>2</sup> et de contrôler la diffusion des références. Les SMO semblent souffrir des inconvénients des applications réparties : chaque SMO implémente une partie de la politique. Si on regarde un système avec des SMO, il est difficile de se rendre compte de la politique globale du système. C'est pour résoudre ce problème que nous allons introduire la notion de *domaines virtuels* (cf. [RH98b]). Un domaine virtuel va permettre de définir les propriétés globales du système.

Un domaine virtuel est constitué d'objets et de références. Seuls les objets d'un domaine sont capables d'utiliser les références sur des objets du même domaine.

Un objet peut avoir plusieurs rôles. Par exemple, un objet du domaine peut agir sous l'identité de différentes personnes suivant le domaine avec lequel il interagit. Les références vers d'autres domaines. Un objet va pouvoir avoir différents rôles, il va par exemple pouvoir agir sous différentes identités quand il interagit avec des objets d'autres domaines. Il va falloir agir avec précaution, les références sont utilisées comme des *capabilities* pour agir sous une autre identité. Si on passe la référence d'un SMO du domaine à un objet à l'extérieur de celui-ci, l'objet extérieur au domaine va pouvoir agir sous l'identité contenu dans le SMO.

Dans certains cas, il peut être utile de déléguer ses droits mais ce n'est pas vrai dans le cas général. Les SMO sont capables de garder une trace des références passées, on va utiliser cette fonctionnalité pour

---

2. utilisateurs,sujets

supprimer les *principal SMO*<sup>3</sup> quand les références quittent le domaine. On a besoin d'être sûr que les SMO vont être invoqués quand les références sortent du domaine. La seule façon de faire cela est d'avoir des SMO attachés à toutes les références qui référencent d'autres domaines. La figure 2.7 montre un état invalide du système : un SMO est sorti de son domaine virtuel. Symétriquement, on va appliquer la même technique à toutes les références qui pointent vers des objets à l'intérieur du domaine.

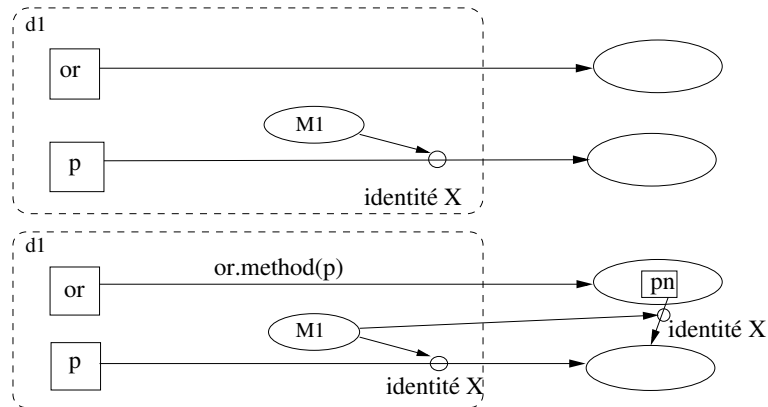


FIG. 2.7 – Sortie du domaine virtuel d'un SMO<sub>p</sub>

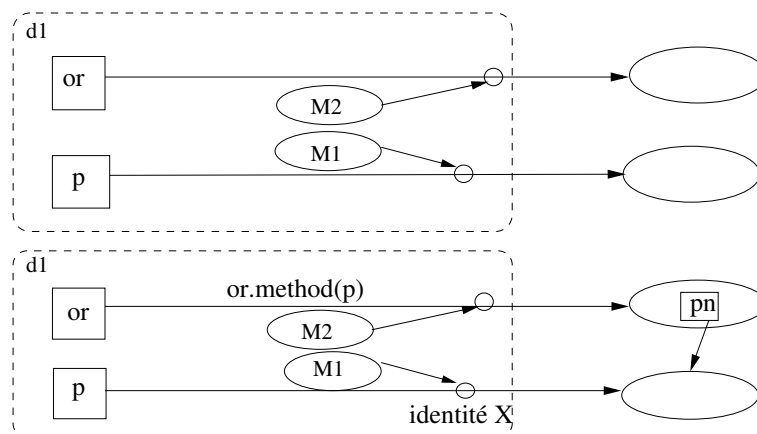


FIG. 2.8 – Suppression automatique des SMOs à la sortie d'un domaine

La figure 2.8 montre un état sécurisé. Les références *or* et *p* sont protégées par des SMO. Quand le paramètre *p* va être passé, il est d'abord intercepté par le SMO *m2* qui vérifie les droits et supprime les SMO du domaine qui sont attachés à cette référence.

On va distinguer deux sortes de SMO, les SMO *source attachment* et les SMO *destination attachment* voir figure 2.10 page 16. Les premiers vont fournir les informations sur les *principals*<sup>4</sup>. Les deux sortes conservent une trace des références échangées. Si on passe des paramètres via des SMO source ou si on retourne des valeurs via des références via des SMO destination cela veut dire que ces références vont quitter le domaine. Si les paramètres passés via références sur lesquels sont attachés des SMO destination ou si on retourne des valeurs par des références avec des SMO source attachés alors ces références vont entrer dans le domaine. voir figure

Ces domaines sont organisés de façon récursive, on va pouvoir inclure les domaines les uns dans les autres. Un attachement est constitué d'un type d'attachement (*src,dst*) et d'un meta-objet. Une référence consiste en une liste d'attachements et un objet cible. Comme exemple, on peut considérer comme référence la liste suivante ((*src,m1*)(*src,m2*)(*dst,m3*)).

3. SMO qui permet d'agir sous une certaine identité

4. utilisateurs

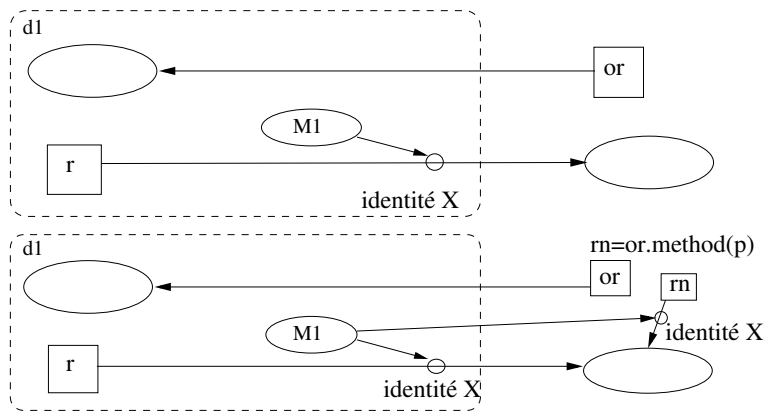


FIG. 2.9 – Les références sans SMO pointant vers l'intérieur d'un domaine sont dangereuses

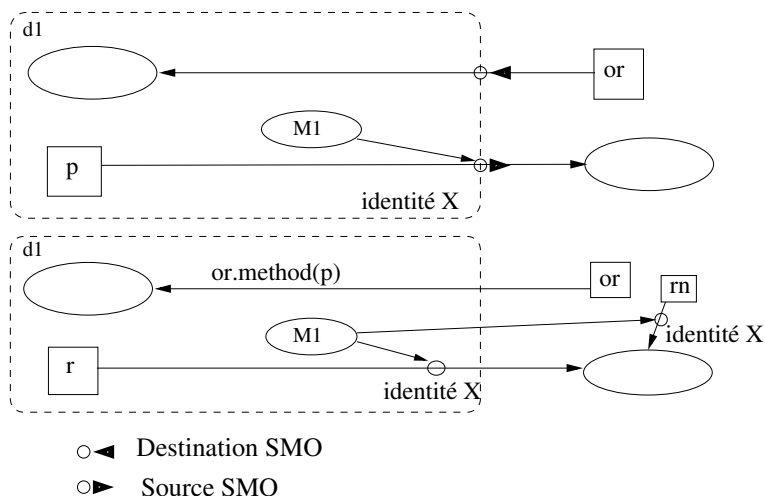


FIG. 2.10 – Les références sans SMO pointant vers l'intérieur d'un domaine sont dangereuses

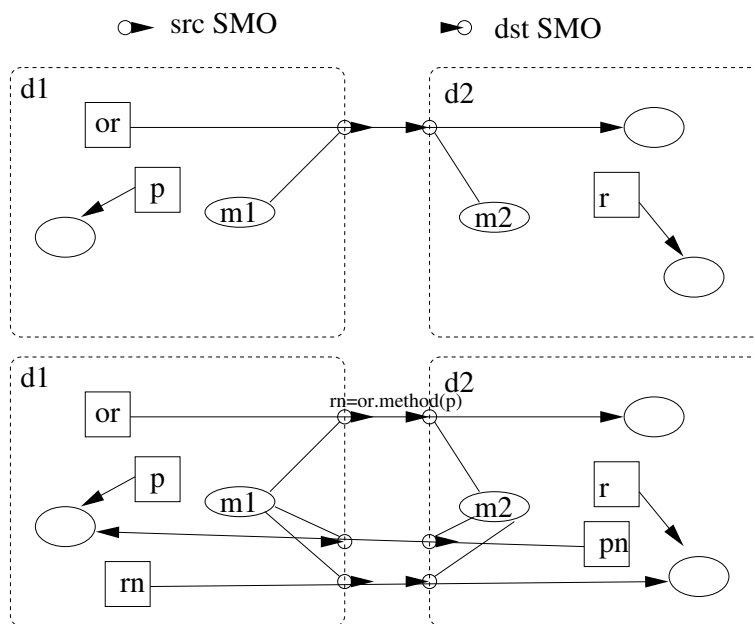


FIG. 2.11 – Invocation de méthode sur un autre domaine

Chaque méta-objet est assigné à un domaine. Il ne peut être utilisé que par ce domaine. Une référence n'est valide dans un domaine, ainsi seul les objets qui se trouvent dans le domaine peuvent utiliser cette référence pour faire des appels de méthode. Une référence ne peut être passée à un objet d'un autre domaine sans être modifiée.

Dans [RH98a], Riechmann et Kleinoder ont démontré de façon formelle leur théorie sur les *Security Meta Objects*.

## Conclusion

Cette étude introduit un modèle de sécurité basé sur des *Security Meta Objects*. Il permet une configuration fine du système d'authentification. L'ajout de domaines virtuels aide à la définition globale de la politique de sécurité du système. Ce modèle permet de prévenir la propagation involontaire des références et l'utilisation de privilèges. Pour leur approche, les auteurs ont modifié la machine virtuelle java afin qu'elle puisse capturer tous les appels de méthode. Nous n'utiliserons pas cette méthode car la librairie ProActive se base sur une machine virtuelle java standard. Il existe cependant d'autres méthodes permettant d'intercepter les requêtes sans avoir à modifier le fonctionnement de la machine virtuelle.

### 2.2.2 Jini

#### Présentation de Jini

L'architecture Jini a pour but de fédérer un ensemble de périphériques, de composants logiciels dans une seule infrastructure distribuée sur le réseau et dynamiquement configurable. La technologie Jini a été développée pour faciliter l'assemblage, le désassemblage et la gestion de tout type de réseau composé de services et des clients utilisant ces services.

Jini permet à un utilisateur de découvrir et d'utiliser avec facilité des services sur un réseau. Pour ce faire, Jini s'appuie sur des fonctionnalités standard de Java, tels le RMI Remote Method Invocation qui permet la programmation distribuée, ainsi que sur des fonctionnalités propres, tels le leasing, qui permet à l'infrastructure sous-jacente de s'adapter aux changements de configuration du réseau.

Le concept principal dans Jini est le concept de service. Il est complété par celui de bail qui permet de garantir à un client l'accès à un service durant une période négociée dans le protocole du service. Le concept de transaction est présent dans Jini ainsi que celui d'événement.

#### La sécurité

Traditionnellement, la sécurité est basée sur un système d'authentification et sur des listes de contrôles d'accès locales *ACL*. C'est le cas dans la plupart des systèmes distribués. Cependant cette approche a de nombreux inconvénients comme par exemple le problème de la protection des opérations de gestion des listes de contrôles d'accès. Dans [BFK99], les auteurs s'accordent à dire que l'utilisation d'un système d'identification à clés publiques avec des ACL est une solution inadaptée aux problèmes de sécurité dans des systèmes réparties.

Une solution alternative, appelée *trust management*, est d'utiliser un ensemble de mécanismes unifiés pour spécifier à la fois la politique de sécurité et les autorisations. Pour cela, on utilise habituellement des certificats pour décrire les actions autorisées pour certains *principals*<sup>5</sup>. On peut citer comme système à base de *trust management* le Policy-Maker [BFS97] qui est à l'origine de ce type de système ou le protocole Simple Public Key Infrastructure (SPKI) [EFL<sup>+</sup>98]

Dans la plupart des environnements distribués, un modèle de sécurité centralisé ne peut être envisagé. Dans un système centralisé, l'authentification se fait par le biais d'une tierce partie de confiance qui doit toujours être en fonction. On peut aussi rejeter les solutions qui utilisent une tierce partie de confiance pour signer le code d'une application. On peut ainsi connaître l'auteur du code. Ce genre de solution est

---

5. utilisateurs

plus intéressante pour reprocher un comportement inattendu au concepteur d'une application que pour protéger le système des éventuelles faiblesses du code.

Le système d'autorisation par chaînage de certificats comme SPKI semble plus approprié à gérer des systèmes décentralisés. Voyons comment marche généralement un système à base de certificats. Le service des clés délivrent toutes les permissions pour un service à un administrateur défini par sa clé qui pourra à son tour permettre aux utilisateurs ordinaires l'accès au service. L'autorisation est donnée sous la forme d'un certificat dans lequel l'administrateur délègue les droits d'accès à la clé publique d'un utilisateur. Ce certificat est conservé du côté du client. Par la suite, l'utilisateur peut déléguer un sous-ensemble de ses droits d'accès aux applications qu'il veut exécuter. Cette délégation des droits d'accès peut varier en fonction du niveau de confiance que l'utilisateur accorde à l'application. L'utilisateur doit aussi veiller à ne pas accorder trop de droits à une application qui n'en a pas besoin.

### 2.2.3 Ajanta

#### Présentation

Ajanta [KT98, TKV<sup>+</sup>99] est un système à base d'agents mobiles. Un agent mobile est un objet qui peut de manière autonome décider de migrer vers un système distribué afin d'y accomplir des tâches sous l'identité de leur créateur. Chaque nœud du réseau qui supporte les agents mobiles doit fournir un serveur d'agents, c'est-à-dire un programme qui permet d'accueillir les agents.

Les agents sont des objets Java qui possèdent la particularité de pouvoir migrer et de posséder leur propre thread d'exécution. La mobilité est implémentée à l'aide de la sérialisation java qui permet de capturer l'état courant de l'objet, de le transmettre sur un autre serveur et de recréer l'objet sur le nouveau serveur. L'agent s'exécute sous l'identité d'une personne à laquelle il appartient. On parle de son *propriétaire*. Cet agent a été créé par une application ou un autre agent, on parle alors de cette entité comme de son *créateur*.

#### modèle de sécurité

Chaque agent emporte avec lui un certificat qui lui a été donné par son propriétaire. Ce certificat contient le nom de l'agent, le nom de son créateur, celui de son propriétaire, celui de son gardien. Il est signé avec la clé privée de son propriétaire. Le protocole Agent Transfer Protocol (ATP) défini par Ajanta a la charge du transport sécurisé des agents. Ajanta fournit aussi un protocole d'authentification utilisé pour toutes les interactions client-serveur. Le *Name service*<sup>6</sup> d'Ajanta permet à toutes les entités d'y enregistrer leurs clés.

Afin de protéger ses ressources, le serveur ne laisse pas l'agent accéder directement à la ressource qu'il demande mais il lui passe un proxy chargé de vérifier systématiquement l'identité de l'appelant et ses autorisations.

Afin de se protéger, un agent possède plusieurs mécanismes. Un agent possède quelques objets en lecture seule. Pour éviter qu'il soit modifié, il va les signer. Cela doit se faire à la création de l'agent avec la clé privée du créateur. Il suffira d'utiliser la clé publique diffusée grâce au Name Service pour vérifier l'intégrité des objets. Ensuite, un agent est généralement envoyé sur des serveurs afin d'y collecter des informations. On va vouloir protéger ces informations de toute modification. Pour cela, on va chiffrer récursivement les différentes valeurs avec la localisation de l'agent qui a transmis l'objet et la localisation du serveur sur lequel se trouvait l'agent, signé par le serveur avec sa clé privée. Au retour de l'agent sur son hôte, on va pouvoir déchiffrer récursivement les objets en utilisant la clé privée de l'agent et la clé publique du serveur sur lequel il se trouvait.

---

6. Service de Nom

### 2.2.4 Modèle de sécurité

#### Abstractions

Dans [Car99], *Cardelli* identifie les problèmes qui se posent dès qu'on introduit la notion de mobilité sur des réseaux grande échelle.

Il décrit trois phénomènes observables sur de grands réseaux, dont un concerne directement la sécurité :

**La localisation virtuelle** : Du fait de la présence potentielle d'agents malveillants, des barrières doivent être érigées entre deux domaines administratifs mutuellement suspicieux. Un programme doit être en mesure de connaître sa localisation, sa provenance, les moyens qu'il doit utiliser pour communiquer ou se déplacer entre deux domaines différents. Cette notion de domaines administratifs séparés introduit la notion de localisation virtuelle et de distances virtuelles.

Cardelli met en évidence deux types de mobilité :

- mobilité logicielle (mobile computation ou mobile software) : un même composant logiciel peut changer de domaine physique ou logique. (ex: un agent qui change de domaine)
- mobilité matérielle (mobile computing ou mobile hardware) : un même composant matériel peut changer d'emplacement physique ou logique, (ex: déplacement d'une portable)

On peut remarquer que la mobilité logicielle peut avoir lieu avec des moyens physiques et que la mobilité matérielle peut avoir lieu avec des moyens logiques :

- un agent logiciel peut se déplacer sur un réseau d'un domaine à un autre mais peut-être également déplacé physiquement en même temps que l'ordinateur sur lequel il est hébergé. Quand l'agent arrive sur un domaine, il va devoir subir des tests de sécurité afin de savoir si ce déplacement était autorisé.
- des solutions logicielles permettent de prendre le contrôle d'un ordinateur à distance. Ceci revient à déplacer physiquement l'ordinateur

#### Le modèle des Ambiants

**ambient** : un ambient est un endroit délimité par une frontière où sont effectués des calculs. Tout ambient possède un nom, une collection de processus locaux et une collection de sous-ambients.

Exemple d'ambients :

- une page web délimitée par un fichier.
- un espace de nommage délimité par un intervalle d'adresses (DNS).
- un système de fichiers délimité par un volume.
- un objet délimité par lui-même.

Les ambients peuvent être hébergés chez d'autres ambients et forment ainsi des arbres logiques. Les ambients permettent donc de définir des domaines hiérarchiques.

Une métaphore possible pour les ambients serait de les assimiler à un répertoire contenant lui-même des sous-répertoires.

#### Sécurité dans le modèle des ambients

Pour effectuer des opérations de base, les ambients doivent y être explicitement autorisés. Pour cela, Cardelli définit trois types de permissions liées à la sécurité :

- permission d'entrer dans le domaine  $n$ .
- permission de sortir du domaine  $n$ .
- permission d'exécuter des processus dans le domaine  $n$ .

Un ambient possède un secret, *son nom*, qui lui permet de s'authentifier auprès des autres ambients. Le secret propre à chaque ambient peut être utilisé pour générer des clés de session. Ces clés serviront au chiffrement des communications entre les différents ambients. Cardelli suppose dans son modèle que les problèmes liés à l'établissement et à la distribution des clés ont déjà été résolus.

## Le modèle de sécurité de Puliafito et Tomarchio

Dans [PT00], les auteurs proposent un modèle de sécurité pour les agents mobiles. Ce modèle est moins théorique que celui de Cardelli. Il a d'ailleurs été implémenté dans le cadre du projet Mobile Agent Platform (MAP).

Ce modèle concerne la migration. Il définit des mécanismes pour protéger :

- les hôtes des attaques d'agents malicieux.
- les agents des attaques des autres agents.

Ce modèle permet de faire du contrôle d'accès, de l'authentification et de la vérification d'intégrité. Les flux ne sont pas chiffrés. Le modèle utilise des ACL et des List Capabilities.

Un domaine est un ensemble non vide d'hôtes. On va pouvoir définir sur ce modèle deux types de politiques de sécurité :

- pour les migrations inter-domaines.
- pour les migrations intra-domaines.

On définit aussi trois niveaux d'authentification :

- authentification de l'auteur du code (personne physique).
- authentification de l'utilisateur de l'agent. La notion d'utilisateur n'a de sens qu'à l'intérieur d'un domaine.
- authentification d'un domaine lorsqu'il s'agit de migrations inter-domaines.

L'authentification et l'intégrité utilisent la cryptographie à clés publiques.

La vérification de l'intégrité des agents s'effectue sur le *bytecode* et sur des *états*.

## 2.3 Outils existants

### 2.3.1 Remote Method Invocation et sérialisation

*ProActive* utilise Java Remote Method Invocation (RMI) [Mic00b]. Solution proposée par Sun pour l'invocation distante de méthodes.

Un objet *O* qui implémente l'interface Remote de RMI est en fait constitué de plusieurs objets. Il possède un stub RMI qui sert de proxy à l'objet *O*. Le stub est chargé du *marshalling* des requêtes, il les transforme pour qu'elles puissent être envoyées sur le réseau. Ce stub référence un skeleton qui est la partie symétrique du stub mais qui se trouve sur la même JVM que l'objet *O*. Le skeleton s'occupe de l'*unmarshalling* des requêtes et fait les appels sur l'objet *O*. Une sérialisation normale d'un objet fait une copie profonde de cet objet. C'est-à-dire qu'il va sérialiser aussi tous les objets référencés par cet objet. La sérialisation RMI va regarder si l'objet est un stub RMI est dans ce cas arrêter la sérialisation au stub.

### 2.3.2 Java Authentication and Authorization Service (JAAS)

JAAS [Mica] est une implémentation Java de Sun du Pluggable Authentication Module (PAM) [Sam96]. Elle est destinée à étendre les fonctionnalités de contrôle d'accès de la plate-forme JAVA2. Cette technologie permet de distinguer clairement dans une application les services d'authentification et d'autorisation. Elle définit une interface de programmation qui permet de faire en sorte que le module d'authentification soit totalement indépendant du reste de l'application. Le module d'authentification agit en quelque sorte comme un «plugin» venant se greffer sur l'application. Ainsi, un développeur peut très facilement adapter le mécanisme d'authentification de son application en fonction de ses différentes contraintes (niveau de sécurité, type d'authentification,...).

### 2.3.3 Java Secure Socket Extension (JSSE) et Java Cryptography Extension (JCE)

La librairie JSSE [Micc] fournit des méthodes permettant de sécuriser la communication réseau entre deux applications. On a du cryptage de données, de l'authentification, du contrôle d'intégrité. JSSE agit juste au-dessus de la couche TCP/IP. JSSE s'interface facilement avec RMI

JCE est un package qui permet d'utiliser des outils de cryptographie, de création de clés publiques et privées.

### 2.3.4 Remote Methode Invocation Secure Extension (RMISE)

#### Présentation

En java, on ne peut savoir qui exécute le code. JAAS étend l'architecture sécurité en fournissant des mécanismes d'authentification, contrôle de l'autorisation d'exécution du code, et des permissions sur les objets. RMISE, évolution du protocole RMI, étend cette sécurité aux applications distribuées en fournissant :

- De l'authentification mutuelle entre le client et le serveur pendant les appels distants.
- La protection des communications.
- Une politique d'exécution du code sur le serveur suivant l'identité du client.

RMISE définit une API de haut niveau, l'implémentation des mécanismes de cryptographie et des protocoles utilisés est cachée volontairement afin que le code écrit au travers de cette API soit le plus portable possible.

#### Les extensions

RMISE permet à la fois aux clients et aux serveurs d'exprimer les contraintes de sécurité devant être appliquées aux appels distants.

- Les contraintes de base : intégrité de la communication, authentification du serveur, authentification du client, délégation. Pour la délégation, il est possible de spécifier sa durée de validité.
- Authentification restreinte aux utilisateurs qui ont des instances de certaines classes.
- Authentification restreinte à certains utilisateurs.

les contraintes sont divisées en deux types, les *requirements* et les *preferences*. Une contrainte de type *requirement* doit être satisfaite pour que l'appel distant puisse se faire. Une *préférence* est une contrainte désirée, à satisfaire si possible, et, si elle n'entre pas en conflit avec une contrainte de type *requirement*. Quand deux *preferences* sont en conflit, une des deux est choisie et satisfaite. Le choix se fait de manière totalement arbitraire. Un serveur peut spécifier des contraintes pour chaque appel distant. Un client peut spécifier des contraintes sur un proxy, dans ce cas, ces contraintes vont s'appliquer à tous les appels faits à travers ce proxy, ou le client peut spécifier des contraintes suivant le contexte (la thread) qui vont s'appliquer à tous les appels faits à partir de cette thread.

Les contraintes peuvent venir de 4 sources:

- Les contraintes du serveur attachées à un proxy.
- Les contraintes du client attachées à un proxy.
- Les contraintes du client attachées à la thread héritées de la thread mère.
- Les contraintes spécifiées à travers le bloc d'activation courant.

Ce mécanisme est conçu de telle manière qu'un client ne puisse diminuer le niveau de sécurité mis en place par un serveur.

Un appel sécurisé ne sera exécuté que si à la fois le client et le serveur supportent les *requirements* demandés par l'autre partie.

Une fois que le client a téléchargé le proxy et avant d'appeler une méthode, il faut vérifier l'intégrité du code. Dans le cas général, le serveur va être identifié comme valide par le client, puis le client va demander au serveur s'il authentifie le code. Si le serveur authentifie le proxy et que le client authentifie le serveur, alors le client par transition va authentifier le proxy.

## Chapitre 3

# Un modèle de sécurité pour ProActive

### 3.1 Principes

Nous avons défini les concepts de base liés à la sécurité et montré les modèles existants. Nous allons maintenant montrer le modèle de sécurité que nous avons élaboré pour la librairie ProActive.

La librairie ProActive ne possède pour le moment aucune notion de sécurité. Elle hérite des règles de sécurité Java définies au sein même de la machine virtuelle. Java ne permet de définir que des règles de sécurité locales à la machine virtuelle (par exemple les droits d'accès aux fichiers). Ceci constitue à la fois un avantage car nous pouvons construire notre modèle de sécurité, mais aussi une faiblesse car certaines parties du modèle de ProActive n'ont pas été conçues pour fonctionner avec notre modèle de sécurité.

Nous nous proposons d'utiliser le niveau *méta* de ProActive pour rajouter les éléments nécessaires à la sécurité.

### 3.2 Les Domaines

Nous avons repris et modifié le concept des *domaines virtuels* introduit dans la section 2.2.1

**Définition 1** *Un nœud est défini par une URL qui est composée du nom DNS ou de l'IP de sa machine hôte suivi du nom du nœud. Chaque nœud possède un nom unique sur un hôte donné.*

Exemple de définition d'un Node : //olla.inria.fr/Node1

**Définition 2** *Un domaine est un ensemble d'unités de calcul qui sont représentées par des nœuds.*

Cette définition nous a laissé beaucoup de souplesse dans l'organisation des domaines. Ainsi un domaine correspondra le plus souvent à un ensemble d'ordinateurs (ou JVM), mais il est tout à fait envisageable d'imaginer qu'un même ordinateur (ou JVM) puisse contenir plusieurs domaines.

Un domaine héberge un ensemble d'objets actifs et leurs arborescences respectives d'objets passifs.

Nous avons énoncé les propriétés suivantes :

**Propriété 1** *un Nœud appartient à un domaine et à un seul.*

**Propriété 2** *Tous les objets à l'intérieur d'un domaine sont soumis à la même politique de sécurité.*

**Propriété 3** *La politique de sécurité d'un domaine est définie à sa création et ne peut-être changée tant que des objets se trouvent à l'intérieur du domaine ou d'un de ses sous-domaines.*

---

#### Exemple 3.2.1 Définition d'un domaine

---

```
Domain D1 {
//potto.inria.fr/Node1
//olla.inria.fr/*
}
```

---

Un domaine peut lui-même contenir des sous-domaines (organisation logique).

**Propriété 4** *Un sous-domaine contient un sous-ensemble des nœuds de son domaine père.*

Dans ce cas, le sous-domaine hérite de la politique de sécurité de son père. Il ne peut définir de règles qui affaiblissent cette politique. On va pouvoir avoir une représentation hiérarchique des domaines.

**Définition 3** *Les domaines ne possédant pas de sous-domaines sont appelés des domaines-feuille.*

**Propriété 5** *Les communications entre les objets d'un même domaine-feuille ne sont pas sécurisées.*

**Propriété 6** *Soit un domaine  $\mathcal{D}$  et  $\mathcal{P}(\mathcal{D})$  la politique de sécurité du domaine  $\mathcal{D}$ . Si  $SD$  est un sous-domaine de  $\mathcal{D}$  et  $\mathcal{P}(SD)$  sa politique alors nous avons*

$$\mathcal{P}(\mathcal{D}) \subseteq \mathcal{P}(SD) \quad (3.1)$$

---

### Exemple 3.2.2 Définition d'un sous-domaine

---

```
Domain SD1 SubDomain D1{
//olla.inria.fr/*
}
```

---

### Utilité des sous-domaines

Rien n'interdit que les machines soient réparties sur la planète. Dans ce cas, les communications passeront généralement par le réseau internet et les messages pourront être interceptés.

Prenons comme premier exemple, le cas d'une multinationale qui possède deux usines, une au Japon et l'autre en France. Ces deux usines sont connectés à internet. On va vouloir que ces deux usines appliquent les règles de sécurité définies par l'administrateur pour l'ensemble du domaine de l'entreprise mais aussi qu'elles communiquent entre elles avec un maximum de sécurité. Dans ce cas, l'administrateur va définir la politique globale du domaine, puis créer deux sous-domaines qui vont communiquer avec un maximum de sécurité.

Prenons maintenant le cas d'un ordinateur portable d'une entreprise. Cet ordinateur va être mobile mais veut toujours pouvoir accéder au réseau de son entreprise. Nous pouvons vouloir combiner deux domaines et donc deux politiques de sécurité.

## 3.3 Politiques de sécurité

Elles vont être associées à chaque domaine. Elles vont permettre de définir :

- la politique de sécurité à appliquer lors de communications vers d'autres domaines
- les différentes permissions associées à un domaine.

Chaque Nœud du domaine possède un objet qui va gérer la politique du domaine. Par communications vers d'autres domaines, on entend l'envoi de requêtes, l'envoi de réponses, la migration d'objets actifs. Les communications sont toujours point-à-point avec rendez-vous.

Nous n'avons pas souhaité séparer explicitement les politiques de sécurité inter et intra domaines.

### Permissions

**L'envoi et la réception de requêtes** permet de savoir si les objets se trouvant dans le Domaine courant peuvent émettre ou recevoir des requêtes vers un autre domaine.

**L'envoi et la réception de réponses** permet de savoir si les objets du domaine courant sont autorisés à accepter ou envoyer des réponses provenant d'un autre domaine.

**L'autorisation des migrations** permet de savoir si les objets actifs présent dans le domaine sont autorisés à migrer.

**La création d'objets actifs** permet de savoir si on autorise la création d'objets actifs dans le domaine courant.

De cet ensemble de permissions, on peut émettre quelques remarques.

**Remarque 1** *Accepter la réception de requêtes venant d'objets du domaine  $D$  implique l'autorisation de pouvoir répondre aux objets de ce domaine  $D$  sous peine de blocage des applications.*

**Remarque 2** *De même, accepter l'émission de requêtes vers les objets d'un domaine  $D$  implique l'autorisation d'accepter des réponses des objets du domaine  $D$  sous peine de blocage des applications.*

**Remarque 3** *Les permissions associées à la réception des requêtes d'un domaine peuvent être différentes des permissions associées aux réponses renvoyées à ce même domaine.*

### 3.4 Modes de communications

Nous avons défini quatre modes de communication entre les domaines.

**Authentification** : Les entités doivent s'authentifier avant de communiquer, c'est-à-dire qu'elles doivent être en mesure de prouver leur identité aux autres entités. L'authentification peut être *unilatérale* ou mutuelle.

**Confidentié** : les communications entre deux entités sont illisibles par d'autres entités.

**Intégrité** : Les entités sont assurées de l'intégrité de leur communication. Elles ont la garantie que les messages qu'elles reçoivent ou émettent sont bien conformes aux messages initiaux.

**Anonymat** : Une entité peut ne pas vouloir divulger son identité.

A chaque mode de communication nous associons une des trois priorités suivantes :

- obligatoire
- optionnel
- interdit

**Propriété 7** *Si une priorité obligatoire ou interdite n'est pas satisfaite, la communication est refusée.*

**Propriété 8** *Une priorité optionnelle est à satisfaire si possible et si elle n'entre pas en conflit avec une priorité obligatoire ou interdite.*

Nous avons conçu un *parser* capable d'interpréter les fichiers de politiques de sécurité des différents domaines et de communiquer à l'application des représentations de ces politiques sous la forme d'objets. Les mécanismes de sécurité sont instanciés automatiquement en fonction des politiques.

#### Conflits

Les conflits surviennent quand deux domaines n'ont pas les mêmes politiques de sécurité. Cela survient quand les politiques des deux domaines n'ont pas été écrites par une même personne.

Deux domaines désirant communiquer n'ont a priori pas forcément deux politiques de sécurité compatibles. Cette incompatibilité va générer un conflit. Ces conflits peuvent être de deux types :

- résolvable : on va adapter les politiques de sécurité afin d'en trouver une commune. Pour fédérer ces deux politiques, nous allons utiliser l'approche de l'interopération<sup>1</sup> afin de conserver les propriétés initiales. Voir l'exemple 3.4.2.
- irrésolvable : les deux politiques sont incompatibles car leur fédération amènerait à des états impossibles. Voir l'exemple 3.4.3.

---

1. voir section 1.4

---

**Exemple 3.4.1** Règles de sécurité du domaine D1

---

```
#Authentification obligatoire : A+
#Authentification interdite   : A-
#Confidentialité obligatoire  : C+
#Confidentialité interdite    : C-
#Intégrité obligatoire       : I+
#Intégrité interdite         : I-
#Anonymat obligatoire        : Z+
#Anonymat interdit          : Z-
```

```
# format des permissions [A,C,I,Z]
#si un permission est optionnelle,
#on ne l'écrit pas
```

```
#exemple pour l'envoi de requêtes
#du domaine D1 vers le domaine D2
#où on demande authentification,
#confidentialité, intégrité et où
#on refuse l'anonymat
```

```
D1 -> D2 : [A+,C+,I+,Z-]
```

---



---

**Exemple 3.4.2** conflit résolvable des règles de sécurité entre deux domaines

---

```
#Politique du domaine D1
D1 -> D2 : [A+,C+,I+,Z-]
```

```
#Politique du domaine D2
D1 -> D2 : [A+,C+,Z-]
```

```
#politique résultante acceptable par les 2
# [A+,C+,I+,Z-]
```

---



---

**Exemple 3.4.3** conflit non résolvable des règles de sécurité entre deux domaines

---

```
#Politique du domaine D1
D1 -> D2 : [A+,C+,I+,Z-]
```

```
#Politique du domaine D2
D1 -> D2 : [A-,C-,I-,Z-]
```

```
#Politique résultante : impossible
# [A--,C--,I--,Z-+]
```

---

Soit un l'anonymat alors que celui avec lequel il désire communiquer peut demander une authentification. Dans ce cas, nous levons des exceptions et les communications ne peuvent pas avoir lieu. Il appartient donc aux concepteurs de l'application distribuée de s'assurer que les définitions de leurs politiques de sécurité ne présentent aucun conflit.

Lorsque deux politiques de sécurité ne sont à priori pas compatibles, nous essayons de voir s'il n'y a pas de compromis qui puisse satisfaire les deux politiques. Par exemple, si un domaine exige l'intégrité mais n'est pas contre la confidentialité, et que l'autre domaine exige la confidentialité, nous forçons le mode confidentiel tout en conservant l'intégrité. Nous avons classé les différents modes par niveau de priorité. Ainsi :

Normal ;Authentification ;Intégrité ;Confidentialité ;Anonymat (la proposition A ;B signifie B est prioritaire sur A.)

Ainsi si  $M(A) ; M(B)$  ( $M(A)$  désigne le mode de A) nous utilisons le mode  $M(A) \cup M(B)$  en s'assurant au préalable que  $M(A)$  n'interdit pas  $M(B)$  (et réciproquement que  $M(B)$  n'interdit pas  $M(A)$ ).

Nous avons défini une grammaire (voir Annexe A, page 34) qui permet de définir la composition des domaines et la politique de sécurité associée à chaque domaine.

### 3.5 Infrastructure à clé publique

Nous avons intégré dans Proactive une infrastructure à clés publiques basée sur une *Tierce Partie de Confiance*.

La tierce partie de confiance peut-être définie comme une autorité de sécurité, à laquelle d'autres entités accordent leur confiance en matière d'activités relatives à la sécurité. Par définition elle est reconnue de confiance par ses adhérents. Ceci est fondamental, car il ne doit pas être possible de remettre en cause son honnêteté et son intégrité.

Le rôle de la tierce partie de confiance intégrée dans ProActive est de garantir l'intégrité et l'authenticité de la clé publique d'une entité, en liant de façon sûre la valeur de la clé à ses caractéristiques. Pour cette raison nous l'appellerons autorité de certification de clés publiques .

La finalité de l'autorité de certification est d'émettre un bloc de données, signé, normalisé, qui crée le lien entre l'identité de l'entité et sa clé publique. Une fois réalisé, ce certificat devient le *passport électronique* de l'entité qui peut alors se présenter auprès des autres entités comme étant accréditée par l'autorité de certification.

Pour générer un certificat, l'autorité de certification crée pour chaque entité une paire de clés publique-privée. L'entité conserve secrètement la clé privée car elle lui servira à signer ses messages. Elle peut en revanche diffuser librement sa clé publique car elle permet à d'autres entités de vérifier sa signature. De plus, les autres entités peuvent avec cette clé publique chiffrer des messages que seule l'entité pourra déchiffrer avec sa clé privée. Le schéma 3.1, page 26 récapitule le principe de certification :

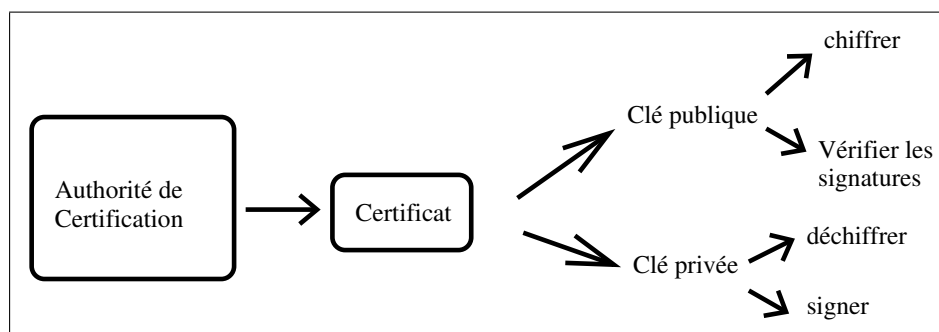


FIG. 3.1 – Principe de certification

### Certificats utilisés dans ProActive

Les différents formats de certificats actuellement reconnus sont nombreux (X.509, PKCS (Public Key Cryptography Standards) de RSA Inc., SPKI (Simple Public-Key Infrastructure) de l'IETF, PGP (Pretty Good Privacy), ...). Ils ne sont hélas pas du tout adaptés à ProActive car ils ont été définis dans le but de certifier les identités d'individus (adresse, email, numéro de téléphone, ...) et non pas les identités des Nodes ou des domaines. Nous nous sommes donc fortement inspirés de ces différents standards pour définir un nouveau type de certificat très adapté à ProActive.

Le format des certificats utilisés dans ProActive est composé de deux parties :

- Un certificat public que le Node peut diffuser librement.
- Un certificat privé que le Node ne doit en aucun cas diffuser.

Le certificat public contient l'URL du Node, le nom du domaine auquel il appartient, sa clé publique et un chaîne de caractères permettant de donner une description de ce Node, le tout étant signé par l'Autorité de Certification.

Le certificat privé contient lui une copie du certificat public ainsi que la clé privée du Node.

Nos certificats ont l'originalité d'être de véritables objets signés, et à notre connaissance, aucun autre système informatique n'intègre de tels certificats. Ce format s'accommode très bien avec les communications de ProActive, qui rappelons le, sont basées sur des échanges d'objets. Nous avons fait en sorte que ces certificats soient sérialisables pour qu'il puissent être sauvegardés dans des fichiers comme les certificats standards.

## Chapitre 4

# Implémentation avec des Protocoles à Méta-Objets (MOP)

### 4.1 Création des objets actifs sécurisés

Pour pouvoir être actif, un objet doit implémenter l'interface `Active`. Cette interface définit les noms des classes des objets à utiliser comme `ProxyForBody` et `Body`. Nous avons simplement étendu l'interface `Active` en un nouvelle interface `SecureActive`. Cette nouvelle interface modifie les noms des classes des objets `ProxyForBody` et `Body` en `SecureProxyForBody` et `SecureBody`.

---

**Exemple 4.1.1** Définition de l'interface `SecureActive`

---

```
package fr.inria.proactive.security;

public interface SecureActive extends fr.inria.proactive.Active
{
    /**
     * The name of the default proxy class used for reified instances of classes
     * implementing Active.
     */
    public static String PROXY_CLASS_NAME = SecureProxyForBody.class.getName();

    /**
     * The name of the default body class used for active instances of classes
     * implementing Active.
     */
    public static String BODY_CLASS_NAME = SecureBody.class.getName();
}
```

---

Comme le montre l'exemple 4.1.1, à part le fait que l'objet doit implémenter l'interface `SecureActive`.

### 4.2 Création des domaines

D'après la définition que nous avons donné des domaines, ils sont constitués de `Nodes`. En effet, un `Node` est un endroit où on va pouvoir faire migrer des objets actifs. C'est l'endroit où les objets actifs vont pouvoir exécuter leur code.

Il suffit de spécialiser les `Nodes` en leur rajoutant des fonctionnalités liées à la sécurité. Un `Node` va pouvoir maintenant vérifier sa politique locale avant d'accepter la création ou la migration d'un objet actif.

---

**Exemple 4.1.2** Un objet actif normal et un objet actif sécurisé

---

```
import fr.inria.proactive.*;
import fr.inria.proactive.security.*;

public class A implements Active {

    public String string = null;

    public A() {}

    public A(String s) {
        this.string = s;
    }

    public String toString() {
        return "A says : " + string;
    }
}

public class SecureA implements SecureActive {

    public String string = null;

    public SecureA() {}

    public SecureA(String s) {
        this.string = s;
    }

    public String toString() {
        return "SecureA says : " + string;
    }
}
```

---

### 4.3 Sécurisation des appels

Dans le cas des appels de méthodes, retour de paramètres, cryptages des informations, ce sont les méta-objet ProxyForBody et Body qui vont être spécialisés. Le schéma 2.1 à la page 10 va être ainsi légèrement modifié, comme le montre le schéma 4.1 à la page 30.

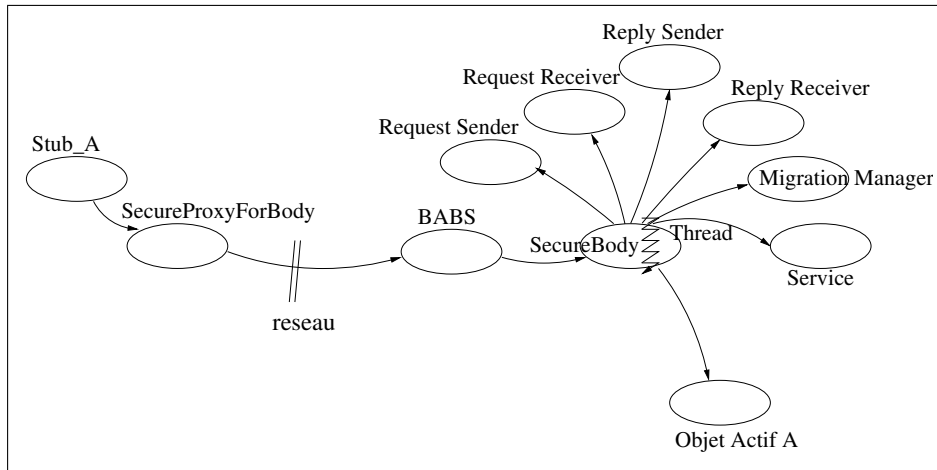


FIG. 4.1 – Schéma d'un objet actif gérant la sécurité

#### Cas des objets actifs

Dans le cas d'un appel de méthode sur un objet actif, nous avons un stub qui est chargé de faire passer l'appel de méthode aux méta-objets. Ces méta-objets, le ProxyForBody dans ce cas, vont avoir la charge de vérifier la politique de sécurité locale et autoriser ou non l'appel de méthode.

#### Appel sur des objets passifs

Dans le cas d'un appel de méthode sur un objet passif, il n'y a pas comme pour un objet actif de méta-objet capable d'intercepter l'appel de méthode. Nous ne pouvons pas sécuriser les appels sur les objets passifs. Mais les objets passifs appartiennent à un objet actif. Cet objet actif est le point d'entrée obligatoire pour atteindre les objets passifs qui se trouvent dans son sous-système. C'est ce dernier qui va contrôler l'accès à ses objets passifs.

#### Négociation

Le ProxyForBody va demander la politique à appliquer vers le domaine sur lequel se trouve le Body de l'objet actif. Si ce domaine n'est pas autorisé l'appel est interdit et une exception est levée. Sinon on communique au Body les contraintes de sécurité demandées. Le Body reçoit les flags, il regarde dans sa politique de sécurité si les communications avec le domaine appelant sont autorisées. Si elles ne le sont pas, une exception est levée et retournée par le réseau sinon l'appel est fait.

On peut imaginer de passer une référence ProActive avec un Proxy modifié. Cette référence serait limitée dans le temps ou dans le nombre d'appels autorisés sur une méthode. Il suffit de spécialiser le Proxy pour qu'il soit capable de gérer des durée de validité des références ou des jetons.

#### Cryptage

Les échanges entre le Proxy et le Body se font au moyen de messages. Les messages sont divisés en deux groupes:

- Les messages de type Request qui vont du ProxyForBody vers le Body qui est en fait l'appel de méthode.

- Les messages de type Reply qui vont du Body vers le ProxyForBody qui contient les paramètres de retour.

Pour chiffrer ces messages, nous avons deux solutions :

La première consiste à utiliser JSSE. Cette solution a l'avantage d'être déjà prête à l'emploi mais possède un inconvénient majeur : le temps de mise en place du flux sécurisé. Pour sécuriser le flux, il faut que les deux parties échangent des informations et définissent des clés de session. cette phase d'initialisation requiert un minimum de 10 secondes. Ce temps est énorme par rapport à un appel de méthode sur un objet actif qui ne dure en moyenne que 2 ms. C'est le temps de cette phase d'initialisation qui nous a fait arrêter les tests que nous faisons.

Nous avons ainsi testé une autre solution, l'encryptage des objets Request et Reply. Cette méthode a l'avantage de nous laisser libre du choix des mécanismes de cryptages à utiliser.

Cependant, on ne peut pas simplement sérialiser la requête, la crypter et l'envoyer par RMI sur le réseau. En effet, si on crypte la requête et qu'on la passe ensuite à RMI, ce dernier ne pourra pas modifier les références RMI qui s'y trouvent car elles sont cryptées. Une solution serait de pouvoir faire sérialiser la requête par RMI afin qu'il modifie les références RMI, de la récupérer, de la crypter et enfin de l'envoyer. Mais il n'existe aucune méthode pour faire cela. Pour le moment, les objets RMI de la requête ne sont donc pas cryptés.

## 4.4 Benchmarks

La tableau 4.1, page 31 nous donne les différentes durée des protocoles liés à la sécurité:

génération de certificat	7000 ms
vérification de certificat	200 ms
génération de clé de session	6000 ms
authentification bilatérale	3000 ms

TAB. 4.1 – Durées d'établissements des protocoles

L'authentification et la génération de la clé de session ne se font pas à chaque appel mais à chaque fois qu'un clé de session à besoin d'être établie. La clé de session a une durée de validité limitée. Cette durée est paramétrable, elle peut aller de 1h à 24h. Lorsqu'une clé de session arrive à expiration, elle n'est pas renouvelée immédiatement. Une nouvelle clé sera créée lors du prochain appel de méthode

Le tableau 4.2 page 32 compare les temps mis par les méthodes suivant l'utilisation du mode sécurisé ou non de ProActive. Les tests ont été effectués en changeant les paramètres des requêtes. On a tout d'abord testé une méthode ne prenant pas de paramètres, puis prenant un entier, un long, une chaîne de caractères et un objet actif. On s'aperçoit que le mode sécurisé de ProActive entraîne un surcoût de l'appel de méthode distant. Ce surcoût est d'environ un facteur 7. Mais il n'intervient que lors de l'appel de méthode distante et le retour des résultats. La durée d'exécution de la méthode n'est pas modifiée.

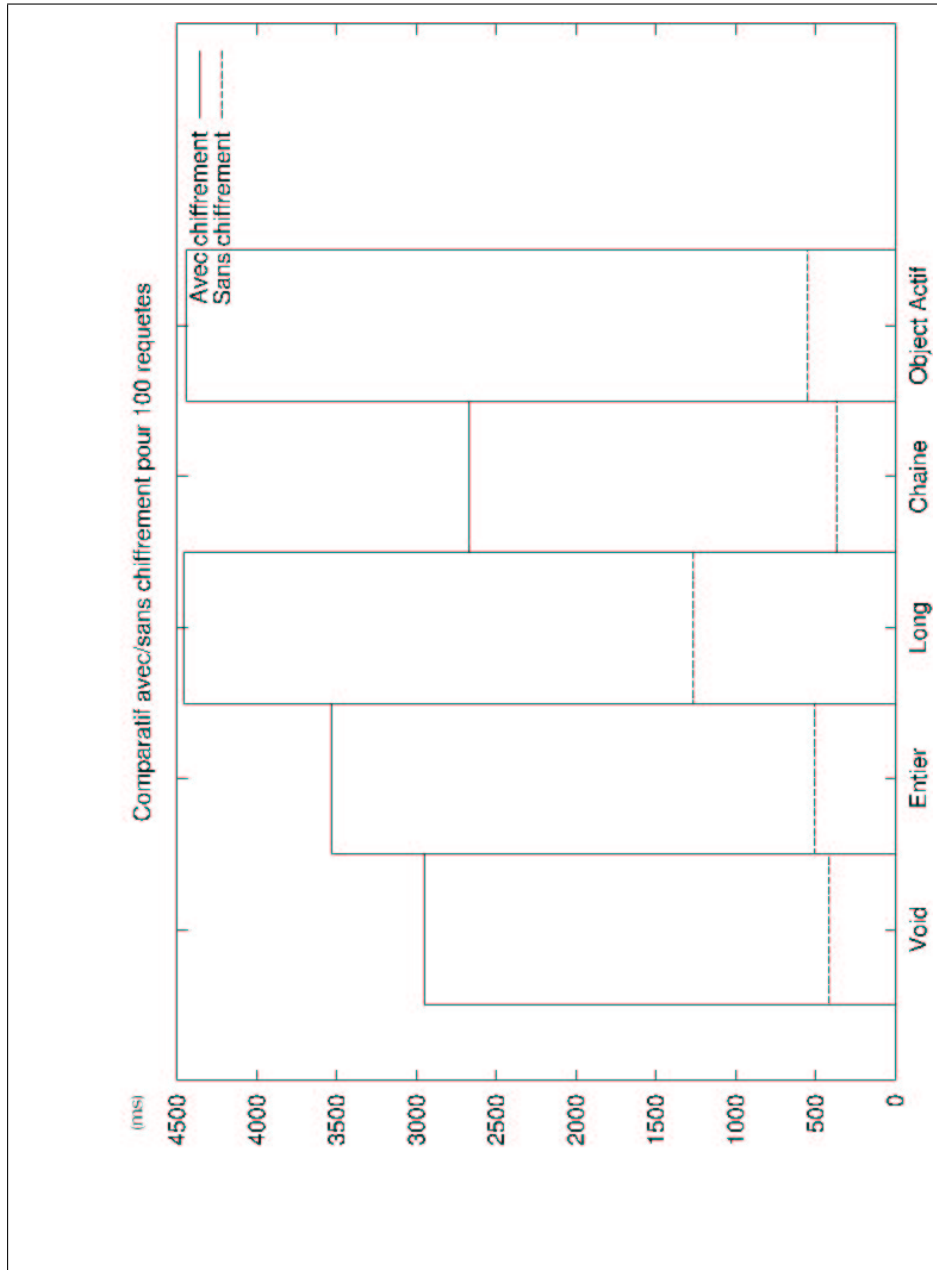


FIG. 4.2 – Surcoût du chiffrement, temps en ms

---

## Chapitre 5

### Conclusion

Le but de ce stage était de proposer un modèle de sécurité pour une bibliothèque permettant la création d'applications parallèles, distribuées et concurrentes.

#### Bilan

Nous avons commencé par faire un tour d'horizon des problèmes liés à la sécurité dans les applications distribuées et introduire quelques notions élémentaires de cryptographie. Nous avons, ensuite, présenté la librairie ProActive et fait un état de l'art des solutions existantes dans le domaine des applications distribuées. De cette étude, nous avons proposé notre propre modèle de sécurité pour la bibliothèque ProActive. Notre modèle propose de mettre en place des domaines qui vont accueillir les objets actifs et de définir des politiques de sécurité entre ces domaines. Regrouper les Nodes qui ont les mêmes politiques de sécurité en un domaine est utile pour permettre d'avoir une vision rapide du système. Nous permettons une organisation hiérarchique des domaines. Cette fonctionnalité est essentielle pour définir une politique globale cohérente d'un système distribué.

Finalement, pour valider notre modèle de sécurité, nous l'avons inclus avec succès au sein de la bibliothèque ProActive.

#### Perspectives

Pour le moment, les seuls sujets sur lesquels porte la sécurité sont les Domaines. On pourrait étendre cette notion de sujet aux utilisateurs. Les autorisations elles-aussi ne se basent que sur le domaine de provenance de l'appel. On peut réduire le degré de granularité des autorisations jusqu'au niveau de la méthode. Pour le moment les applications n'ont pas accès aux primitives permettant de gérer la sécurité, seuls les méta-objets peuvent y accéder. On pourrait les laisser accéder à un sous-ensemble de ces fonctions afin que l'application puisse gérer son modèle de sécurité. Les Nodes actuellement ne possèdent moyen de contrôler l'exécution des objets qu'ils hébergent. Un objet va pouvoir utiliser toutes les ressources (mémoire, temps CPU) de la machine hôte. Il semble nécessaire de rajouter un moyen de contrôler ces probables débordements. Sumatra [ARS97] propose déjà ces fonctionnalités, elle repose sur une machine virtuelle modifiée.

Les forwarders qui font partie intégrante de ProActive posent de nombreux problèmes au niveau de la sécurité. Par exemple, une requête provenant d'un domaine D va suivre le chemin indiqué par les forwarder jusqu'à l'objet cible. L'objet cible va traiter la requête, et va vouloir renvoyer le résultat. Si la politique de sécurité du domaine dans il se trouve l'autorise, cela ne pose pas de problème. Au contraire, si l'envoi de réponse vers le domaine D n'est pas autorisé, un problème va se poser. Une solution pourrait être de lever une exception et de la communiquer à l'application distante.

L'intégration de la sécurité aux autres axes de recherches de ProActive, notamment dans les communications de groupe. Un groupe peut avoir besoin d'identifier les objets qui veulent en faire partie, de communiquer de façon confidentielle avec tous ses membres ou utiliser les fonctions d'intégrité pour être que chaque membre recevra un message qui n'aura pas été modifié.

## Annexe A

## Grammaire de la politique de sécurité

```

DomainName = String |
            "*" # all domains
DomainDef  = "Domain" DomainName { NodeSet } |
            "Domain" DomainName "Subset" DomainName { NodeSet }

NodeSet    = Node+
Node       = "/" Machine "/" String
DNS_Name   = String |
            "*" "." String "." String
HostName   = String | String "." String "." String
IP         = IPNumber "." IPNumber "." IPNumber "." IPNumber |
            IPNumber "." IPNumber "." IPNumber "*" |
            IPNumber "." IPNumber ".*" |
            IPNumber ".*.*" |
            "*"
Machine    = HostName | IP
String     = Alpha String | Number String
Alpha      = [a-z,A-Z]
IPNumber   = [0-255]
Number     = [0-9]

RULES      = PERMISSION* MIGRATION*
PERMISSION = DomainName UNI_BI DomainName ":" COM_MODE
MIGRATION  = DomainName "migration" FROM_TO DomainName : YES_NO
COM_MODE   = [ MODE_A , MODE_I , MODE_C , MODE_Z ]
Option_Mode = Option Mode |
            NULL

# authenticate
MODE_A     = A Option |
# integrity
MODE_I     = I Option |
# confidentiality
MODE_C     = C Option |
# anonymous
MODE_Z     = Z Option |
Option     = + | -
UNI_BI     = "->" | # Query
            "<->" | # Query & Reply
            "<-" | # Reply
FROM_TO    = "->" |
            "<-"

```

```
YES_NO      = "YES" |  
             "NO"
```

## Bibliographie

- [Co91] Commission of the European Communities . Information technology security evaluation criteria (ITSEC), version 1.2, 1991.
- [Ale99] D. Alexander. Security in active networks, 1999.
- [ARS97] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.
- [BFK99] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.
- [BFS97] M. Blaze, J. Feigenbaum, and M. Strauss. Ieee symposium on security and privacy, 1997.
- [Bir97] Kenneth P. Birman. Building secure and reliable network applications. In *WWCA*, pages 15–28, 1997.
- [BS93] E. Biham and A. Shamir. *A Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993.
- [Car97] Luca Cardelli. Mobile Computations. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 3–6. Springer-Verlag: Heidelberg, Germany, 1997.
- [Car99] Luca Cardelli. Abstractions for mobile computation. In *Secure Internet Programming*, pages 51–94, 1999.
- [CG97] L. Cardelli and A. Gordon. A calculus of mobile ambients, 1997.
- [CGA98] L. Cardelli, A. Gordon, and M. Ambients. Foundations of software science and computational structures, 1998.
- [CHV00] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple proxy-based mop with security awareness, 2000.
- [Com] N. Communications. Secure socket layer reference document.
- [Dil94] A. Diller. An introduction to formal methods, 1994.
- [EFL+98] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory. Internet Draft, 1998.
- [ELZN00] P. Eronen, J. Lehtinen, J. Zitting, and P. Nikander. Extending jini with decentralized trust management, 2000.
- [EN01] P. Eronen and P. Nikander. Decentralized jini security, 2001.
- [FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
- [FKK96] A. Freier, P. Karton, and P. Kocher. Secure socket layer, 1996.
- [Fol92] S. Foley. Aggregation and separation as noninterference properties, 1992.
- [Ger98] E. Gerck. Overview of certification systems: X, 1998.
- [GGKL89] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 1989 National Computer Security Conference*, 1989.
- [GQ96] Li Gong and Xiaolei Qian. Computational issues in secure interoperation. *Software Engineering*, 22(1):43–52, 1996.
- [Hos92] H. Hosmer. Metapolicies ii, 1992.

- [IH] Naomaru Itoi and Peter Honeyman. Pluggable authentication modules for Windows NT. pages 97–108.
- [Inc99a] S. Inc. Java remote method invocation security extension, 1999.
- [Inc99b] S. Inc. Jini architecture specification – revision, 1999.
- [KdR91] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.
- [KG96] J. Kleinder and M. Golm. Metajava: An efficient run-time meta architecture for java, 1996.
- [KT98] Neeran Karnik and Anand Tripathi. Agent Server Architecture for the Ajanta Mobile-Agent System. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, 1998.
- [KT99] Neeran Karnik and Anand Tripathi. Security in the ajanta mobile agent system. Technical report, 1999.
- [LV00] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. In *Public Key Cryptography*, pages 446–465, 2000.
- [McL88] J. McLean. The algebra of security. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1988.
- [McL96] John McLean. A general theory of composition for a class of “possibilistic” properties. *Software Engineering*, 22(1):53–67, 1996.
- [Mica] S. Microsystems. Java authentication and authorization service (jaas).
- [Micb] S. Microsystems. Java cryptography extension (jce).
- [Micc] S. Microsystems. Java secure socket extension (jsse).
- [Mic98] S. Microsystems. Java object serialization specification, 1998.
- [Mic00a] S. Microsystems. Java specification request 76: Rmi security, 2000.
- [Mic00b] S. Microsystems. Remote methode invocation, 2000.
- [Mtf90] D. McCullough, A. theorem, and f security. *Ieee transactions on software engineering*, 1990.
- [oT98] N. of and S. Technology. Advance encryption standard development effort webpage, 1998.
- [PHN00] M. Phillipson, B. Haumacher, and C. Nester. More efficient serialization and rmi for java, 2000.
- [PT00] A. Puliafito and O. Tomarchio. Security mechanisms for the map agent system, 2000.
- [RH98a] Thomas Riechmann and Franz J. Hauck. Meta objects for access control: a formal model for role-based principals. In *Workshop on New Security Paradigms*, pages 30–38, 1998.
- [RH98b] Thomas Riechmann and Franz J. Hauck. Meta objects for access control: extending capability-based security. In *Proceedings of the ACM New Security Paradigms Workshop*, pages 17–22, New York, NY, 1998. ACM Press.
- [RK98] Thomas Riechmann and Jurgen Kleinoder. Meta objects for access control: Role-based principals. In *Australasian Conference on Information Security and Privacy*, pages 296–307, 1998.
- [Sam96] Vipin Samar. Unified login with pluggable authentication modules (pam). In *ACM Conference on Computer and Communications Security*, pages 1–10, 1996.
- [Sec] R. Security. Public key cryptography standards : Pkcs.
- [SS88] B. Clifford Neuman J. G. Steiner and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988.
- [TKV<sup>+</sup>99] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. A mobile agent programming system, 1999.
- [VJ99] Jan Vitek and Christian D. Jensen. *Secure Internet programming: security issues for mobile and distributed objects*, volume 1603. Springer-Verlag Inc., New York, NY, USA, 1999.
- [Wal98] J. Waldo. Jini architecture overview, 1998.
- [WS98] Ian Welch and Robert J. Stroud. Dynamic adaptation of the security properties of applications and components. In *ECOOOP Workshops*, page 282, 1998.